VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Výkonný hybridní minipočítač pro robotický operační systém

Hybrid Minicomputer for Robotic Operating System

2019 Bc. Daniel Trnka

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Daniel Trnka**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Výkonný hybridní minipočítač pro robotický operační systém
Hybrid Minicomputer for Robotic Operating System

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je vybrat vhodný minipočítač na platformě iMX8M a naprogramovat potřebné programové vybavení tak, aby minipočítač mohl sloužit pro řízení robotického systému s využitím ROS (Robotického operačního systému) v OS Linux. Funkcionalita ROS musí být rozšířena o využití procesorového jádra ARM Cortex-M4 pro řízení časově kritických událostí.

1. Seznamte se s rodinou procesorů iMX8M a vyberte vhodný dostupný minipočítač, jehož celkové technické parametry budou lepší než v současnosti pro ROS běžně používaný minipočítač Beaglebone Black.
2. Upravte jádro OS Linux tak, aby bylo možno z OS Linux měnit program jádra Cortex-M4 a nenarušit spojitost komunikace mezi jádry.
3. Připravte pro jádro M4 příklady použití dostupných periférií. Při návrhu zohledněte možnost předávání dat z/do OS Linux.
4. Rozšiřte ROS o moduly umožňující ovládání dostupných periférií a to jak s OS Linux, tak pro časově kritické události pomocí jádra M4.
5. Pro prezentaci nových možností počítače navrhněte vhodné ukázkové aplikace integrované do ROS.
6. Otestujte spolehlivost navržených řešení, časové odezvy a datovou propustnost mezi jádry.

Seznam doporučené odborné literatury:

[1] Minipočítač PICO-PI-8M: https://www.wandboard.org/products/picopiimx8m/PICO-PI-IMX8M-PRO/
[2] Procesorová rodina iMX8: https://www.nxp.com/imx8
[3] Aleš Prchal, Řízení robotické ruky pomocí vícejádrového hybridního procesoru, Diplomová práce, VŠB-TUO, FEI, katedra informatiky, 2017

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Petr Olivka, Ph.D.**

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2019 .....

Rád bych na tomto místě poděkoval Ing. Petru Olivkovi, Ph.D. za odborné vedení při tvorbě této práce a za všechny předané zkušenosti během studia.

Abstrakt

Cílem diplomové práce je vybrat cenově dostupný minipočítač malých rozměrů s hybridním procesorem i.MX 8M, který obsahuje dedikované jádro pro real-time aplikace. Pro vybraný minipočítač je v rámci práce připravená distribuce spolu s upraveným linuxovým jádrem, které obsahuje podporu pro jaderný framework remoteproc. Framework umožňuje nahrávat nový kód na dedikované jádro za běhu systému a při startu dedikovaného jádra reinicializuje VirtIO struktury tak, aby byly v konzistentním stavu. Dále je vysvětlena problematika ochrany paměti a souběžného používání periférií z obou typu jáder. Pro jednotlivé periférie jsou připraveny ukázky použití jak na dedikovaném jádře, tak také v operačním systému Linux. Komunikace mezi robotickým operačním systémem a dedikovaným jádrem je zajištěna pomocí protokolu *rosserial*. Vybraný protokol umožňuje jednoduše využívat funkcionalitu robotického operačního systému přímo na dedikovaném jádře. Na závěr jsou demonstrovány složitější ukázky integrované do robotického operačního systému, na kterých byly otestovány časové odezvy a spolehlivost navržených řešení.

Klíčová slova: i.MX 8M, remoteproc, RPMsg, Robot Operating System, ROS, Linux, FreeRTOS, hybridní procesor, real-time

Abstract

The goal of thesis is to select a small size and low-cost minicomputer with a hybrid processor i.MX 8M, that includes a dedicated core for real-time applications. For the selected minicomputer is created Linux distribution with a modified kernel, that supports loading new code to the dedicated core at runtime via the remoteproc framework. The modified Linux kernel also reinitializes VirtIO structures, which is required for the consistent state of VirtIO at every start of a program on the dedicated core. The issues of concurrent use of peripherals between cores as well as memory protection are also described. Implemented examples demonstrate how to use peripherals from the dedicated core and Linux. The communication between the Robot Operating System and the dedicated core is provided by the *rosserial* protocol. Thanks to this protocol it is easy to use the Robot Operating System directly on the dedicated core. Finally, more complex examples are implemented to demonstrate the use of the dedicated core with the Robot Operating System. These examples were also used to test the overall reliability and response time.

Key Words: i.MX 8M, remoteproc, RPMsg, Robot Operating System, ROS, Linux, FreeRTOS, hybrid processor, real-time

Obsah

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
CPU	–	Central Processing Unit
eMMC	–	Embedded Multimedia Card
FPU	–	Floating-Point Unit
FTP	–	File Transfer Protocol
GPIO	–	General-Purpose Input/Output
GPU	–	Graphics Processing Unit
IOMUX	–	Input/Output multiplexor
IP	–	Internet Protocol
JTAG	–	Joint Test Action Group
MAC	–	Medium Access Control
MDIO	–	Management Data Input/Output
MPU	–	Memory Protection Unit
MU	–	Messaging Unit
NFS	–	Network File System
OS	–	Operating System
POSIX	–	Portable Operating System Interface
PWM	–	Pulse Width Modulation
RAM	–	Random Access Memory
RDC	–	Resource Domain Controller
ROS	–	Robot Operating System
SDK	–	Software Development Kit
SoC	–	System On a Chip
SOM	–	System On a Module
SPI	–	Serial Peripheral Interface
TFTP	–	Trivial File Transfer Protocol
I^2C	–	Inter-Integrated Circuit
UART	–	Universal Asynchronous Receiver-Transmitter
USB	–	Universal Serial Bus

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Hlavním cílem této práce je vybrat vhodný minipočítač s hybridním procesorem i.MX 8M pro robotický operační systém (ROS). Snahou je tak připravit výkonnější vícejádrovou alternativu k minipočítači BeagleBone Black, který je často využíván pro aplikace ROS. Dalším cílem práce je, aby časově kritický kód byl vykonáván na dedikovaném mikropočítačovém jádře a výpočetně náročnější kód byl vykonáván na aplikačních jádrech.

Real-time zpracovávání je potřeba u aplikací vyžadujících odezvu do určitého časového limitu či přesné časování během řízení. Nedodržení může vést k nefunkčnosti nebo k fatálním následkům. Běžné operační systémy nejsou schopné dodržet přesné časování, už jen kvůli své složitosti. Jedním z řešení je použít dva oddělené procesory propojené komunikačním kanálem, nebo využít hybridní procesor, který kombinuje jádra různých typů v jediném pouzdře. Kromě menších rozměrů je výhodou možnost přenášet data mezi jádry efektivněji bez nutnosti využít komunikační periférie a možnost využít integrované periférie na konkrétních jádrech dle potřeby.

Ve druhé kapitole se bude práce zaobírat výběrem cenově dostupného minipočítače malých rozměrů s co nejdelší dostupností na trhu. Vybraný minipočítač bude následně detailněji popsán.

V další kapitole bude pro minipočítač uzpůsobena linuxová distribuce vhodná pro použití s robotickým operačním systémem.

Ve čtvrté kapitole bude rozebrán dosavadní postup nahrávání kódu na dedikované jádro a princip komunikace se systémem Linux. Dosavadní přístup bude analyzován z pohledu vývoje, kdy je potřeba provádět časté nahrávání nového kódu. Případné úpravy budou muset být provedeny na úrovní jádra operačního systému.

V další kapitole budou popsány jednotlivé periférie a jak je použít na M4 jádře, ale také z operačního systému Linux. Protože má dedikované jádro přístup k periferiím, tak lze očekávat možné problémy souběhu, které mohou vést až k nefunkčnosti aplikace. Jedna z podkapitol bude také věnována ochraně paměti, bez které může docházet k hůře laditelným problémům, jako je například nechtěná změna v odeslaných paketech nebo také k bezpečnostním problémům jako eskalace oprávnění.

V šesté kapitole bude popsán robotický operační systém a naimplementována podpora pro komunikaci s M4 jádrem. Cílem je vytvořit co nejjednodušší použití ROS funkcionality přímo na M4 jádře.

Na závěr budou naimplementovány a popsány složitější ukázky pro budoucí použití např. na samořiditelných autíčkách. Ukázky budou sloužit jako prostředek testování spolehlivosti navržených řešení a časové odezvy.

2 Výběr minipočítače

Na trhu je dostupné větší množství minipočítačů, ale jen malá část z nich obsahuje hybridní procesor s dedikovanými jádry pro obsluhu periférií v reálném čase.

Před výběrem minipočítače je nejprve nutné vzít v úvahu hardwarové požadavky pro aplikaci využívající robotický operační systém. Název robotický operační systém může vyvolávat dojem, že bude potřeba nějaká podpora pro vybraný hardware. Přestože ROS má v názvu „operační systém“, tak se jedná o pouhý userspace framework a nejsou zde kladeny žádné speciální požadavky na hardware. Je pouze nutné daný framework provozovat v operačním systému Linux či experimentálně ve Windows či Mac OS. Mezi hlavní výhody patří standardizovaný mechanismus posílání zpráv mezi uzly, které mohou být propojené pomocí IP sítě. Jednotlivé úkoly se tak mohou rozdělit do separátních procesů, mezi kterými nejsou přímé závislosti. Detailnější ukázky i s popisem budou popsány v pozdější kapitole.

Robotický operační systém se běžně využívá na minipočítači BeagleBone Black[1]. Minipočítač obsahuje procesor AM335x, který má pouhé jedno 32bitové aplikační jádro Cortex-A8 a 512 MB RAM paměti. K dispozici jsou dvě 32bitová RISC jádra nazvaná PRU (programmable real-time unit) pro obsluhu periférií. Každé PRU jádro má k dispozici 8 kB paměti pro kód a 8 kB pro data. Oproti běžnému přístupu, kdy je snaha co nejvíce používat hardwarové periférie, zde programy vytváří potřebné periférie nastavováním hodnot na pinech. PRU jádra běží na frekvenci 200 MHz a přístup k pinům by měl trvat jediný takt [1]. Nevýhodou však může být vlastní architektura RISC jader, pro kterou je nutný překladač dodávány výrobcem.

2.1 Minipočítače s procesory i.MX

Procesory série i.MX vytvořené firmou NXP, dříve známou jako Freescale, jsou zaměřené na multimediální aplikace vyžadující vysoký výpočetní výkon a malou spotřebu energie [2]. Celý procesor je dodávaný na jednom čipu (System on a Chip - SoC) a obsahuje aplikační ARM jádra, video jádra a také volitelně ARM Cortex-M jádra vhodná pro obsluhu periférií v reálném čase. SoC dále obsahuje periférie, menší paměť a další součásti nutné pro jeho běh. Procesory se dělí do několika sérií. Série i.MX RT obsahuje pouze mikropočítačové jádro Cortex-M7 nebo Cortex-M33 [3]. Procesory ze série i.MX 6 až i.MX 8 obsahují primárně aplikační Cortex-A jádra a volitelně mikropočítačová jádra Cortex-M. Shrnutí použitých jader některých vybraných procesorů lze vidět v tabulce 1.

V první sérii i.MX 6 obsahuje procesor i.MX 6SoloX jedno aplikační jádro Cortex-A9 a mikropočítačové jádro Cortex-M4. Tento procesor například využívá minipočítač UDOO NEO.[2] Real-time aplikace se píší v Arduino frameworku a běží nad real-time systémem MQX, ale je také možné využít open source variantu FreeRTOS [4].

[1] `https://beagleboard.org/black`

[2] `https://www.udoo.org/udoo-neo/`

Tabulka 1: Přehled jader na některých procesorech série i.MX

	Aplikační jádra	**Mikropočítačová jádra**
i.MX RT1064		Cortex-M7
i.MX RT600		Cortex-M33
i.MX 6UltraLite	1x Cortex-A7	
i.MX 6Solo	1x Cortex-A9	
i.MX 6SoloX	1x Cortex-A9	Cortex-M4
i.MX 6Quad	4x Cortex-A9	
i.MX 7Solo	1x Cortex-A7	Cortex-M4
i.MX 7Dual	2x Cortex-A7	Cortex-M4
i.MX 8	2x Cortex-A72, 4x Cortex A53	Cortex-M4F
i.MX 8M	4x Cortex A53	Cortex-M4F
i.MX 8M Mini	4x Cortex A53	Cortex-M4F
i.MX 8M Nano	4x Cortex A53	Cortex-M7
i.MX 8X	4x Cortex A35	Cortex-M4F

V druhé generaci i.MX 7 obsahují všechny procesory Cortex-M4 jádro. Procesor i.MX 7Dual obsahuje dvě aplikační jádra.

Ve třetí a aktuálně poslední sérii i.MX 8 je několik procesorů, ale některé z nich nejsou ještě určeny k prodeji. Příkladem je procesor i.MX 8 obsahující dvě skupiny aplikačních jader, mezi kterými může v době běhu přepínat pro snížení spotřeby. Další procesor i.MX 8M Nano obsahuje výkonnější mikropočítačové jádro Cortex-M7.

Dostupný procesor i.MX 8M obsahuje 4 aplikační ARM jádra Cortex-A53 a jedno jádro Cortex-M4F, které oproti Cortex-M4 obsahuje floating-point unit (FPU) pro urychlení výpočtů v pohyblivé řadové čárce jednoduché přesnosti - v programovacích jazycích obvykle označováno jako `float`. Dále obsahuje gigabitový Ethernet, USB 3.0 Type C, MIPI-CSI a GPU s podporou OpenGL/ES 3.1, OpenGL 3.0, Vulkan, OpenCL 1.2. Procesor je také schopný přehrávat video s rozlišením 4K v reálném čase.

Přestože se jedná o SoC obsahující většinu součástí počítače, nelze jej samostatně využít. SoC je nutné mít umístěny na plošném spoji společně s napájením, DDR paměti, eMMC paměti a konektory pro periférie. Někteří výrobci umísťují procesor spolu s pamětmi a Wi-Fi/Bluetooth modulem na System on Module (SOM), ke kterému dále nabízí základní desku s vyvedenými perifériemi a dalšími obvody. Celek se pak nazývá minipočítač.

V následujících podkapitolách budou krátce popsány minipočítače obsahující procesor i.MX 8M a také jejich výhody či nevýhody. Orientační shrnutí popsaných minipočítačů lze nalézt v tabulce 2. Je potřeba brát na vědomí, že uvedena cena minipočítačů je orientační. Ceny nelze přímo porovnávat, protože některé minipočítače neobsahují např. Wi-Fi modul. Počet pinů je také pouze orientační, protože některé z nich plní funkci uzemnění.

Kritéria pro výběr minipočítače byla: dostupnost vyvedených pinů v rozumné formě, malé rozměry, cena a také dlouhá dostupnost na trhu.

Tabulka 2: Shrnutí parametrů popsaných minipočítačů

	Rozměry	**Piny**	**JTAG**	**DDR**	**eMMC**	**Cena**[3]
MCIMX8M-EVK	100 x 100 mm	I2C	ano	4 GB	16 GB	11 300 Kč
EmCraft i.MX 8M	130 x 146 mm	40	ano	1 GB	4 GB	8 000 Kč
phyBOARD-Polaris	100 x 100 mm	60	ano	1 GB	8 GB	4 800 Kč
HummingBoard Pulse	102 x 69 mm	ne	ne	2 GB	8 GB	5 900 Kč
CuBox Pulse	50 x 50 mm			2 GB	8 GB	3 800 Kč
PICO-PI-IMX8M	85 x 56 mm	40	ne	2 GB	8 GB	4 200 Kč

2.2 MCIMX8M-EVK

MCIMX8M-EVK je referenční minipočítač od firmy NXP, který lze vidět na obrázku 1. Minipočítač obsahuje operační paměť o velikosti 4 GB a 16 GB eMMC paměti. Na plošném spoji je také konektor pro gigabitový Ethernet, USB3, Bluetooth, Wi-Fi a také konektor pro kameru a displej. Výhodou je také vyvedený JTAG konektor, který lze využít nejen pro ladění aplikace na M4 jádru, ale také na případné ladění aplikačních jader.

NXP poskytuje dokumentaci[4] včetně detailního schématu celé desky. Podporovanými systémy jsou Android a Yocto Linux. Pro M4 jádro je připravené MCUXpresso SDK spolu s ukázkami použití periférií a použití real-time operačního systému FreeRTOS. Velkou nevýhodou je absence vyvedených pinů - je zde pouze I^2C, u kterého však není dostupný žádný pin pro napájení. Kvůli tomuto nedostatku, celkovým rozměrům a také ceně, nemá minipočítač využití v rámci této práce.

2.3 EmCraft i.MX 8M SOM se základní deskou

Firma EmCraft nabízí SOM, který je na obrázku 2 připojen do základní desky IMX8M-SOM-BSB s 40pinovým konektorem a konektorem pro JTAG.

Výrobce poskytuje detailní schémata a návody, jak zprovoznit například Wi-Fi či I^2C sběrnici[5]. Dále je k dispozici Yocto vrstva s podporou primárně pro multimediální aplikace. Pro M4 jádro je připravená ukázka „*Hello, world!*“ a ukázka komunikace mezi aplikačním a M4 jádrem.

Minipočítač i přes detailní schémata nebyl vybrán nejen z důvodu velikosti základní desky o rozměrech 130×146 mm, ale také z důvodu vysoké ceny.

2.4 phyBOARD-Polaris

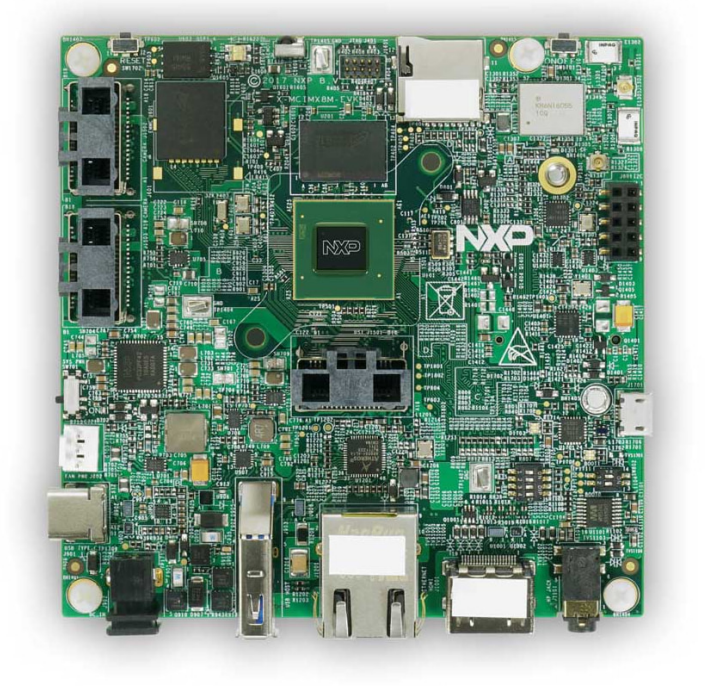
Minipočítač phyBOARD-Polaris[6] se objevil na trhu v průběhu tvorby diplomové práce. Na desce o rozměrech 100×100 mm je 60pinový konektor s GPIO piny včetně rozhraní JTAG.

[3] přepočtená zaokrouhlená cena na stovky nahoru ke dni 29. 3. 2019, z Mouser a stránek výrobců

[4] `https://www.nxp.com/support/developer-resources/run-time-software/i.mx-developer-resources/evaluation-kit-for-the-i.mx-8m-applications-processor:MCIMX8M-EVK?tab=Documentation_Tab`

[5] `https://www.emcraft.com/products/868`

[6] `https://www.phytec.in/product/system-on-modules/phycore-imx-8m/`



Obrázek 1: Minipočítač MCIMX8M-EVK [5]

V době výběru minipočítače nebyla veřejně dostupná jeho cena. Může se však jednat o vhodnou alternativu k minipočítači MCIMX8M-EVK, která obsahuje větší počet vyvedených pinů.

2.5 HummingBoard Pulse a CuBox Pulse

Firma SolidRun nabízí dva minipočítače s i.MX 8M určené spíše pro multimediální aplikace.

Minipočítač HummingBoard Pulse[7] o rozměrech 102×69 mm nemá vyvedené piny v rozumné formě a chybí zde i JTAG konektor. Cena minipočítače s 8 GB eMMC a 2 GB RAM paměti je v době psaní přibližně 5 900 Kč.

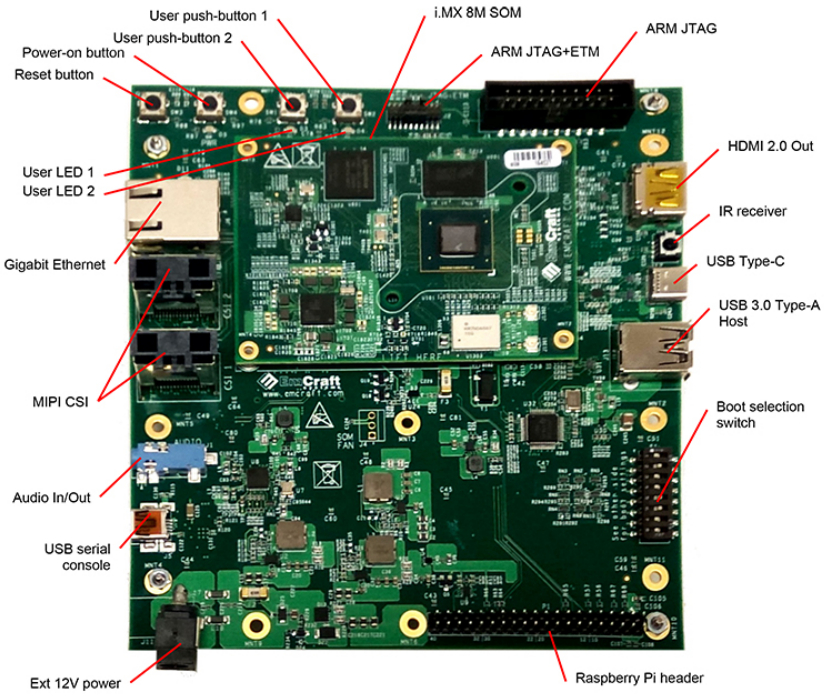
Pro minipočítač CuBox Pulse[8] o rozměrech $50 \times 50 \times 50$ mm nebyly v době tvorby diplomové práce dostupná schémata a pravděpodobně nemá vyvedené žádné piny.

Výhodou obou minipočítačů však může být podpora Power over Ethernet, kdy je možné minipočítač napájet pomocí ethernetového kabelu.

2.6 PICO-PI-IMX8M

Základním prvkem minipočítače na obrázku 3 od firmy TechNexion je SOM nazvaný PICO-IMX8M [7]. Zakoupený modul v předprodeji obsahuje 2 GB DDR a 16 GB eMMC paměti a také modul s Wi-Fi a Bluetooth. SOM modul je připojen třemi konektory na základní desku

[7] `https://developer.solid-run.com/products/hummingboard-pulse/`
[8] `https://developer.solid-run.com/products/cubox-pulse/`



Obrázek 2: Minipočítač EmCraft i.MX 8M [6]

PICO-PI-IMX8M. Minipočítač má stejné rozměry 85×56 mm jako Raspberry Pi a zachovává umístění některých periférií na pinech.

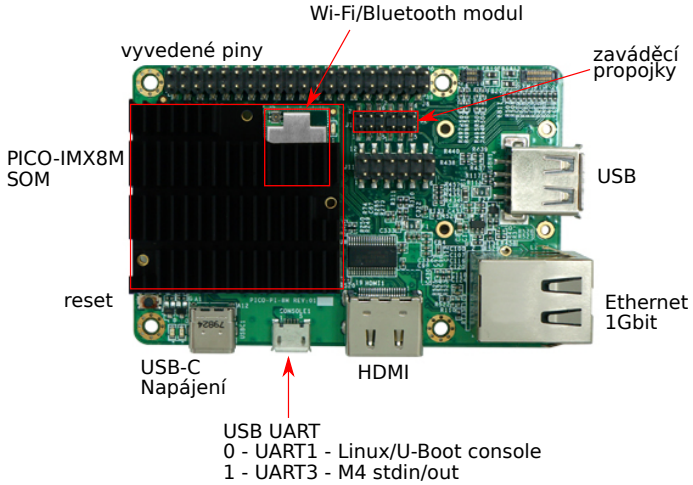
Oproti Raspberry Pi je velkou výhodou gigabitový Ethernet připojený pomocí MDIO a také rychlá integrovaná eMMC paměť. Nevýhodou je absence JTAG konektoru, který už chybí na SOM a nelze tak efektivně ladit aplikace na M4 jádru.

Tento minipočítač nejvíce vyhovoval požadavkům a byl předobjednán ještě pod původním názvem WAND-PI-8M. Minipočítač přišel s přibližně půlročním zpožděním a bez dokumentace. TechNexion zveřejnil linuxové jádro [8] založené na linux-imx verze 4.9.88 od NXP. Dále zveřejnili upravené SDK[9] pro M4 jádro a vrstvu pro Yocto[10]. TechNexion jádro oproti NXP přidává podporu pro Wi-Fi ovladač, různé senzory a také Device Tree konfiguraci pro daný minipočítač. Konkrétní změny lze získat pomocí rozdílu mezi dvěma Git větvemi[11]. V upraveném SDK je hlavně provedena změna, aby standardní vstup či výstup směroval na UART dostupný z microUSB konektoru minipočítače.

2.6.1 Periférie

Protože nebyly zdokumentované vyvedené piny, bylo potřeba vytvořit jejich rozložení. Jednoduchou metodou je připojit LED postupně na jednotlivé vyvedené piny a pokusit se postupně

[9] `https://github.com/TechNexion/freertos-tn/tree/freertos_SDK_2.3_IMX8M`
[10] `https://github.com/TechNexion/meta-edm-bsp-release`
[11] `git diff -name-only imx_4.9.88_2.0.0_ga tn-imx_4.9.88_2.0.0_ga-test`

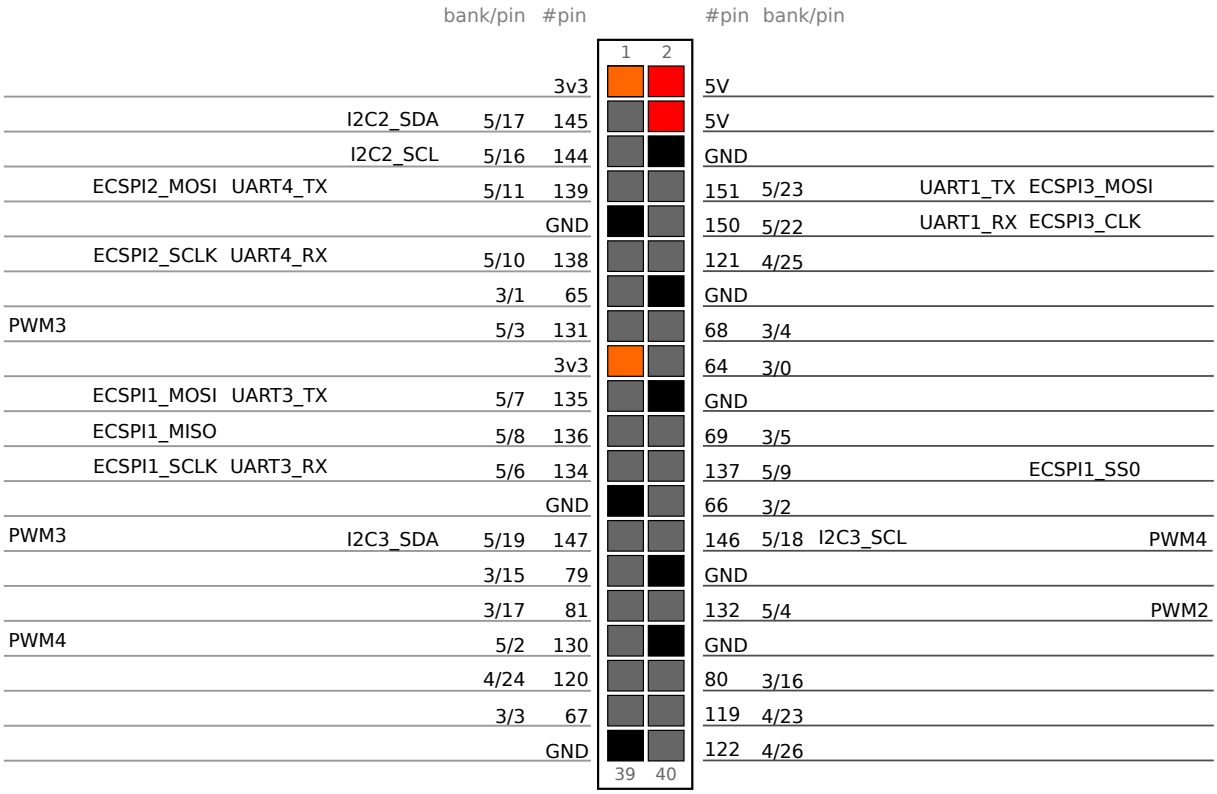


Obrázek 3: Minipočítač PICO-PI-IMX8M

zapsat logickou 1 resp. 0 na všechny GPIO piny. V systému Linux lze provést změnu hodnoty na pinu pomocí již zastaralého GPIO Sysfs rozhraní. U většího počtu pinů se ale stejně neprojevila nastavená hodnota, protože nebyla nakonfigurována GPIO funkcionalita, či bylo nesprávně nastaveno proudové omezení pinů (drive strength field).

Proto piny bylo potřeba otestovat v zavaděči Das U-Boot ještě před zavedením operačního systému, který provádí konfiguraci pinů. Zavaděč Das U-Boot poskytuje příkaz `gpio`, kterým lze nastavit hodnotu pinu. Protože procesor obsahuje 160 pinů a na desce je jich vyvedených 28, bylo by velice zdlouhavé testovat každý pin, zda se LED rozsvítí u všech ze 160 možností. Pro rychlejší nalezení vyvedených pinů byl vytvořen jednoduchý program *gpio-bisect.py*, který pro konkrétní vyvedený pin hledá jeho označení pomocí binárního vyhledávání. Program rozdělí interval možných označení $\langle 0, 160)$ na dvě poloviny. Pro všechny piny v první polovině se nastaví logická jednička. Pokud uživatel zadá, že se připojená LED rozsvítila, tak se první resp. druhý interval rozdělí na dvě poloviny a aplikuje se stejný postup, dokud se nenalezne označení pro daný pin. Každý pin se pomocí tohoto postupu podařilo nalézt do $log_2 160$ kroků. Piny jsou organizovány v portech 1 až 5 po 32 pinech. Číselný pin lze tedy převést pomocí `GPIO((n / 32) + 1)_IO(n % 32)` a následně dohledat alternativní funkce v SoC dokumentaci.

Z vytvořeného rozložení pinů na obrázku 4 lze vidět, že jsou některé periférie jako I^2C, SPI a UART na stejných místech, jako na minipočítači Raspberry Pi. Lze tedy připojit různé rozšiřující desky, ale pravděpodobně nebude dodávaný program fungovat z důvodů jiného číslování pinů či kvůli přímému přístupu k periferiím. Některé rozšiřující desky Raspberry Pi mají na jedné I^2C periférii připojenou paměť, kterou si video core před zavedením systému Linux načte a podle toho dále načte Device Tree overlay pro konfiguraci periférií na rozšiřující desce.



Obrázek 4: Rozložení pinů minipočítače PICO-PI-IMX8M

3 Linuxová distribuce pro minipočítač PICO-PI-IMX8M

Linuxová distribuce se skládá z jádra operačního systému a userspace aplikací a tím tvoří použitelný celek. Nevhodná volba distribuce může zkomplikovat používání, protože například nejsou k dispozici binární balíčky programů či jsou příliš zastaralé. Při absenci balíčků je nutné provádět kompilaci, která může na procesorech s menším výpočetním výkonem trvat dlouho.

V této kapitole je popsána oficiálně podporovaná distribuce a její nevhodnost pro použití spolu s robotickým operačním systémem. Z tohoto důvodu je dále popsána tvorba specifické distribuce přímo pro vybraný minipočítač.

3.1 Oficiálně podporovaná distribuce založená na Yocto

Zpočátku TechNexion poskytoval distribuci vytvořenou pomocí build systému Yocto. Cílem Yocto je standardizovat postup sestavení linuxové distribuce pro různé architektury různých výrobců a umožnit uživatelům tento proces jednoduše opakovat, protože se zajistí mimo jiné i překlad cross-překladačů. Kompletní sestavení ale vyžaduje vysoký výpočetní výkon a větší kapacitu disku.

Yocto je složen z vrstev, které obsahují recepty definující, jak daný program zkompilovat a zabalit. Při překladu systém vyřeší všechny závislosti jednotlivých receptů a pomocí cross-kompilace zkompiluje programy pro cílovou architekturu. Výhodou je také možnost vytvořit vlastní vrstvu upravující chování receptů z jiných vrstev.

Distribuce od TechNexion využívá vrstvy *meta-freescale*[12] od NXP a přidává vrstvu *meta-edm-bsp-release*, která kompiluje jejich upravené jádro či ovladače pro Wi-Fi modul [9].

Pro integraci robotického operačního systému existuje vrstva *meta-ros*[13], která v aktuální době podporuje verzi ROS Indigo, u které končí podpora v dubnu 2019.

ROS poskytuje binární balíčky architektury ARM64 pro distribuci Ubuntu. Při existenci binárních balíčků postrádá Yocto smysl a spíše jen komplikuje přípravu a samotné používání operačního systému, protože při požadavku na další software je nutné znovu provést sestavení celého systému.

Z důvodu zbytečné složitosti a také absence novější verze robotického operačního systému bylo od Yocto upuštěno.

3.2 Vytvoření distribuce založené na Ubuntu

Pro PICO-PI-IMX8M musela být v rámci této práce vytvořena distribuce založena na Ubuntu, protože zde nebyla původně podpora ani ze strany TechNexion ani NXP. Při dokončování práce byla nalezena distribuce Ubuntu verze 18.04 na FTP serveru[14] TechNexion.

[12] `https://github.com/Freescale/meta-freescale`
[13] `https://github.com/bmwcarit/meta-ros`
[14] `ftp://ftp.technexion.net/demo_software/pico-imx8m/`

Před vytvořením distribuce je vhodné naznačit, jak zjednodušeně operační systém startuje. Po startu procesor začne vykonávat instrukce z paměti, kterou lze například zvolit pomocí propojky, přepínače či pojistek. V této paměti je zavaděč, jehož úkolem je inicializovat hardware, jako například nastavit časování DDRAM paměti a následně načíst jádro systému spolu s Device Tree konfigurací do paměti. Device Tree popisuje hardware, aby bylo možné využít stejné jádro s různě připojenými periferiemi či pro různé minipočítače využívající stejnou větev jádra bez nutnosti rekompilace. Po načtení se začne vykonávat kód jádra, který po své vlastní inicializaci připojí root filesystém (rootfs) a spustí první proces nazývaný `init` - v dnešní době se nejčastěji jedná o `systemd`. Další jaderné moduly, které nejsou nezbytně nutné pro inicializaci systému, jsou dostupné na filesystému v adresáři `/lib/modules/verze_jádra/`.

Pro vytvoření distribuce je nejprve vhodné zkompilovat jádro systému Linux společně s Device Tree konfigurací. Protože se ale jedná o kompilaci pro jinou architekturu, je nutné provést tzv. cross-kompilaci, kdy překladač generuje instrukce pro odlišnou architekturu. Nastavení architektury probíhá pomocí proměnné prostředí `ARCH` a nastavení prefixu kompilátoru pomocí `CROSS_COMPILE`. Prefix se také nazývá *target triplet*, který je ve formátu *machine-vendor-operatingsystem* - v tomto případě se jedná o `aarch64-linux-gnu`. Dále je potřeba vytvořit konfiguraci jádra v souboru `.config`. Pro minipočítač je připravena konfigurace, kterou lze zkopírovat do souboru `.config` pomocí `make remoteproc_defconfig`. V konfiguraci se určuje, jaké moduly se zkompilují přímo do jádra či jako dynamický modul. Dále se nastavují parametry modulů, což u ovládačů zařízení není potřeba, protože by měly být definované v Device Tree konfiguraci.

Příkazy nutné pro kompilaci lze vidět ve výpise 1. Po dalších úpravách jádra stačí jen zavolat program `make` pro sestavení změněných částí.

```
$ git clone https://github.com/trnila/linux-tn.git
$ cd linux-tn
$ export CROSS_COMPILE=aarch64-linux-gnu-
$ export ARCH=arm64
$ make remoteproc_defconfig
$ make "-j$(nproc)"
```

Výpis 1: Příkazy pro kompilaci linuxového jádra

Výsledkem je zkompilované linuxové jádro v `arch/arm64/boot/Image` a také zkompilovaná Device Tree konfigurace v `arch/arm64/boot/dts/freescale/*.dtb`. V této fázi jsou také zkompilované dynamické moduly `*.ko`.

Pro minipočítač PICO-PI-IMX8M jsou ze strany TechNexion připravené tři Device Tree konfigurace[15]:

pico-8m.dts Konfigurace rozšiřuje `fsl-imx8mq.dtsi`, která popisuje celý SoC procesor, ale má všechny periferie zakázané. Tato konfigurace potřebné periférie povolí a mimo jiné nakonfiguruje použité piny.

pico-8m-m4.dts Konfigurace rozšiřuje předchozí a zakazuje periférie proto, aby mohly být využité na M4 jádře. V opačném případě by mohlo dojít k nekonzistenci, kdyby systém Linux i M4 aplikace využívala stejné periférie.

pico-8m-dcss-ili9881c.dts Konfigurace zakazuje HDMI výstup a konfiguruje MIPI DSI (Display serial interface) pro připojení LCD řadiče ILI9881C s dotykovou vrstvou komunikující přes I^2C.

V dalším kroku je nutné připravit root filesystém, obsahující základní programy pro běh systému spolu s programem `init`, který je spuštěn jádrem OS jako první proces.

Ubuntu poskytuje základní rootfs[16], ale bez programu `systemd`, což by zapříčinilo zastavení linuxového jádra kvůli neexistenci `init` programu. Tento program je možné změnit v zavaděči na POSIX Shell pomocí parametru jádra (cmdline) `init=/bin/sh`. Po zprovoznění sítě lze následně doinstalovat potřebný balíček s `init` programem a následně restartovat.

Další možnosti je využití programu `debootstrap` a sestavit tak na počítači kompletní distribuci od počátku. Instalace některých balíčků ale vyžaduje vykonání kódu cílovým CPU, což se stává komplikací, pokud je cílová architektura rozdílná jako v tomto případě. Instalaci je nutno provést ve dvou fázích. V první fázi se balíčky stáhnou a rozbalí do cílového adresáře a vytvoří se skript pro druhou fázi, který se provede v emulátoru QEMU. Spouštění virtuálního stroje v QEMU je komplikované, protože je nutno zajistit funkční konektivitu, disk atd. Linuxové jádro poskytuje modul binfmt-misc [10], který programy určené pro jinou architekturu rozliší a spustí v emulátoru. Této funkcionality lze využít společně s programem `chroot`, který změní kořenový adresář vytvářené distribuce a spustí `deboostrap` druhé fáze v emulátoru, a to bez nutnosti vytvářet virtuální stroj.

Přidání robotického operačního systému je triviální, protože je k dispozici repositář s již zkompilovanými balíčky, který stačí přidat a nainstalovat meta-balíček dle potřeby v emulovaném `chroot` prostředí. Základním meta-balíčkem pro použitou ROS verzi melodic je `ros-melodic-ros-base`, který obsahuje pouze nutný základ. Dalšími meta-balíčky jsou `ros-melodic-desktop` a `ros-melodic-desktop-full`, poskytující i grafické aplikace, které je možné provozovat přímo na minipočítači.

Po instalaci všech potřebných balíčků je nutné nainstalovat dynamické moduly do adresáře `/lib/modules/verze_jádra`, které byly zkompilovány během kompilace jádra.

[15] `https://github.com/TechNexion/linux/tree/tn-imx_4.9.88_2.0.0_ga-test/arch/arm64/boot/dts/freescale`

[16] `http://cdimage.ubuntu.com/ubuntu-base/releases/18.04/release/`

V této chvíli je distribuce hotová a připravena k použití. Nástroj k automatizovanému vytvoření distribuce je dostupný v Git repositáři `picopi8m-ros-distbuild`[17]. Skript také nastaví připojení adresáře `/tmp` a logovacích souborů do RAM paměti, aby nedocházelo ke zbytečnému opotřebovávání eMMC paměti během vývoje. Sestavení distribuce také probíhá pomocí průběžné integrace změn[18] při každé nové revizi v GitLab repositáři. Během sestavení se vytvoří nový Docker kontejner *ubuntu:18.10* a spustí celý proces sestavení distribuce. Jedná se zároveň o formu dokumentace a testování, že sestavení funguje v čistém systému bez nutnosti manuálního testování. Výstupem[19] jsou balíčky s linuxovým jádrem, moduly, hlavičkovými soubory a také archiv root filesystému.

3.3 Úprava Device Tree konfigurace

Některé úpravy, jako povolení SPI rozhraní či konfigurace pinů, vyžadují modifikaci v Device Tree konfiguraci. Minipočítač Raspberry Pi řeší úpravy pomocí vrstev nazvaných Device Tree Overlays. Funkcionalita ale není v současném zavaděči minipočítače podporována.

Pro vytvoření vlastní konfigurace je zapotřebí vytvořit nový soubor pojmenovaný například `pico-8m-disable-gpio.dts` v adresáři `arch/arm64/boot/dts/freescale/` ve zdrojových kódech linuxového jádra. Soubor by měl pomocí řádku `#include "pico-8m.dts"` využít již připravenou konfiguraci minipočítače a provést patřičné úpravy. Následně je nutné přidat název tohoto souboru, ale s příponou `.dtb`, do souboru `Makefile` ve stejném adresáři dle výpisu 2. Konfiguraci lze následně zkompilovat pomocí příkazu `make freescale/pico-8m-disable-gpio.dtb`.

```
dtb-$(CONFIG_ARCH_FSL_IMX8MQ) += \
   ...
   pico-8m.dtb \
   pico-8m-m4.dtb \
   pico-8m-disable-gpio.dtb
```

Výpis 2: Úprava Makefile pro kompilaci Device Tree konfigurace

Použití zkompilované konfigurace pak závisí na konkrétní konfiguraci v zavaděči. Ve výchozím nastavení stačí zkompilovanou konfiguraci uložit do adresáře `/boot` a nastavit v zavaděči proměnnou pomocí příkazu `setenv fdt_file pico-8m-disable-gpio.dtb`.

3.4 Zavaděč Das U-Boot

Během upravování jádra či konfigurace je výhodné nahrávat jádro přes síť. Zamezí se tak problému, kdy úprava jádra vede k brzkému pádu celého systému a nemožnosti změnit jádro.

[17] `https://github.com/trnila/picopi8m-ros-distbuild`

[18] `https://gitlab.com/trnila/picopi8m-ros-distbuild/pipelines`

[19] `https://gitlab.com/trnila/picopi8m-ros-distbuild/-/jobs/artifacts/master/raw/picopi-ros.rootfs.tar.xz?job=build`

Použitý zavaděč Das U-Boot lze využít k načtení jádra a konfigurace do paměti pomocí protokolu TFTP nebo NFSv2. Root filesystém lze následně připojit z integrované eMMC paměti nebo protokolem NFS pomoci parametrů:
`root=/dev/nfs nfsroot=$serverip:/srv/nfs/picopi,v3,tcp rw ip=$ipaddr`.
Při úpravách jádra a modulů je také vhodné připojit pomocí NFS adresář `/lib/modules`.

Při opakovaném použití je vhodné si připravit novou „boot položku“ podle následující konvence: do proměnné `bootargs_nazev` se uloží příkaz pro nastavení jaderných parametrů a do `bootcmd_nazev` příkazy pro nahrání jádra s Device Tree konfiguraci. Výchozí boot položku lze poté nastavit do proměnné `bootcmd`. Ukázku pro zavádění jádra a Device Tree konfigurace ze sítě i z eMMC paměti lze vidět ve výpise 3. Nastavení o maximální velikosti 4 kB je uložené v eMMC paměti na pozici `0x1000`[20]. Konfiguraci je také možné zkompilovat pomocí programu `mkimage` do souboru `boot.scr` na prvním FAT32 oddíle eMMC paměti.

```
u-boot=> setenv bootargs_mmc 'setenv bootargs "
  init=/lib/systemd/systemd console=${console}
  root=/dev/mmcblk0p2 rootwait rw cma=${cma_size}"'
u-boot=> setenv bootcmd_mmc '
  run bootargs_mmc;
  fatload mmc 0:1 0x40480000 Image;
  fatload mmc 0:1 0x43000000 ${fdt_file};
  booti 0x40480000 - 0x43000000;'
u-boot=> setenv bootcmd_dev '
  run bootargs_mmc;
  tftp 0x40480000 /dev/kernel/Image;
  tftp 0x43000000 /dev/kernel/${fdt_file};
  booti 0x40480000 - 0x43000000;'
u-boot=> setenv bootcmd 'run bootcmd_mmc'
u-boot=> saveenv
u-boot=> run bootcmd_dev
```

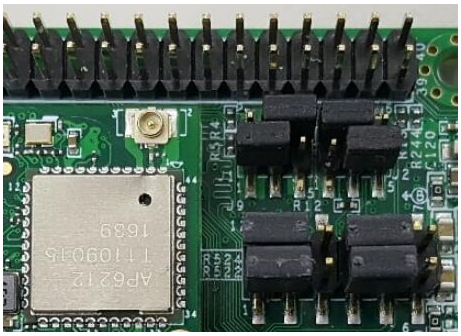
Výpis 3: Příkazy pro konfiguraci zavedení jádra z eMMC paměti nebo protokolem TFTP

3.5 eMMC paměť

Minipočítač obsahuje integrovanou eMMC paměť a oproti paměťovým kartám zde vzniká problém, jak jednoduše nahrát distribuci. Jedním ze způsobů je pomocí propojek dle obrázku 5 nastavit zavádění do režimu *Serial boot*. V tomto režimu je možné do DDRAM minipočítače přes USB-C konektor pomocí programu `MfgTools2` nahrát malou linuxovou distribuci, která po USB poskytne blokové zařízení eMMC paměti. Na počítači následně proběhne zformátování disku na dva oddíly. Na první oddíl se souborovým systémem *FAT32* se nakopíruje jádro systému Linux

[20] `https://github.com/TechNexion/u-boot-edm/commit/44385c93bea216d91b25c046c2c7f1768dc7131a#diff-dee07ccc9988d765c7d1a0265b57f0dcR196`

společně s Device Tree konfigurací. Na druhý *ext4* oddíl se rozbalí root filesystém distribuce. Potřebné příkazy jsou stručně popsány ve výpise 4 a v repositáři `picopi8m-ros-distbuild` je připraven skript `flash.sh`.



(a) Boot z eMMC paměti

(b) Serial boot loader

Obrázek 5: Nastavení propojek pro výběr paměti [11]

```
$ wget ftp://ftp.technexion.net/development_resources/development_tools/
    installer/pico-imx8m_mfgtool_20180911.zip
$ unzip pico-imx8m_mfgtool_20180911.zip
$ cd pico-imx8m_mfgtool_20180911/mfgtools
$ chmod +x mfgtoolcli
# bash ./linux-runvbs.sh mfgtool2-pico_imx8-use_eMMC_as_usb_mass_storage.vbs
# cfdisk /dev/sdX
# mkfs.fat32 /dev/sdX1
# mkfs.ext4 /dev/sdX2
# mount /dev/sdX2 /mnt
# mkdir /mnt/boot
# mount /dev/sdX1 /mnt/boot
# cd /mnt
# curl -L https://gitlab.com/trnila/picopi8m-ros-distbuild/-/jobs/artifacts/
    master/raw/picopi-ros.rootfs.tar.xz?job=build | tar -xJ
```

Výpis 4: Přehled potřebných příkazů pro instalaci distribuce na eMMC paměť

3.6 Wi-Fi

Wi-Fi není součástí SoC a je zde jako samostatný obvod Qualcomm Atheros QCA9377 na System On Module PICO-IMX8M.

Použitá Wi-Fi vyžaduje připojenou externí anténu, out-of-tree linuxový modul a také binární firmware, který nelze volně distribuovat. Firmware je nutné uložit do adresáře `/lib/firmware` před nahráním modulu. Postup pro zprovoznění Wi-Fi a jejího otestování je ve výpise 5. Příkaz `make modules_install` provede instalaci modulu `wlan.ko` do adresáře s ostatními moduly.

Pro trvalou konfiguraci lze použít NetworkManager nebo systemd-networkd společně se službou WPA supplicant.

Wi-Fi a Bluetooth modul je aktivní pokud je na pinu `GPIO3_IO24`[21] resp. `GPIO3_IO21` přivedena logická jednička. V případě zakázaní periférie GPIO v Linuxu, je nutné odpovídající pin pro povolení Wi-Fi nastavit v zavaděči pomocí příkazu `gpio set 88`.

```
$ git clone https://github.com/TechNexion/qcacld-2.0.git -b tn-CNSS.LEA.NRT_3.0
$ cd qcacld-2.0
$ KERNEL_SRC=/lib/modules/$(uname -r)/build CONFIG_CLD_HL_SDIO_CORE=y
    CONFIG_PER_VDEV_TX_DESC_POOL=1 SAP_AUTH_OFFLOAD=1 CONFIG_QCA_LL_TX_FLOW_CT
    =1 CONFIG_WLAN_FEATURE_FILS=y CONFIG_FEATURE_COEX_PTA_CONFIG_ENABLE=y
    CONFIG_QCA_SUPPORT_TXRX_DRIVER_TCP_DEL_ACK=y
    CONFIG_WLAN_WAPI_MODE_11AC_DISABLE=y TARGET_BUILD_VARIANT=user
    CONFIG_NON_QC_PLATFORM=y CONFIG_HDD_WLAN_WAIT_TIME=10000 make -j4
$ make modules_install
$ depmod -a
$ modprobe wlan

$ ip link set wlan0 up
$ iw dev wlan0 scan
$ wpa_passphrase BSID password > wpa_supplicant.conf
$ wpa_supplicant -i wlan0 -c wpa_supplicant.conf
$ dhclient wlan0
$ ping 1.1.1.1
$ ping 2600::
```

Výpis 5: Postup pro zprovoznění a otestování Wi-Fi

[21] https://github.com/TechNexion/linux/blob/3a31571795c513e1516e5e9d67f7c997f1ddee99/arch/arm64/boot/dts/freescale/pico-8m.dts#L145

4 Nahrávání kódu na M4 jádro a komunikace s Linuxem

V této kapitole je nejprve popsán dosavadní postup nahrávání kódu na M4 jádro a následná komunikace se systémem Linux. Dosavadní přístup je spíše cílen na produkční nasazení, kdy je potřeba nahrát nový kód pouze při startu minipočítače. Z tohoto důvodu je naimplementován a popsán způsob, který umožní při vývoji restartovat M4 jádro bez narušení komunikace mezi jádry. Informace uvedené v této kapitole jsou platné pro SoC i.MX 8M a tedy nezávislé na zvoleném minipočítači.

4.1 Dosavadní postup

V této podkapitole je popsán aktuální postup kompilace a nahrávání kódu na M4 jádro. Posléze je popsán i způsob komunikace mezi jádry a problém, který se vyskytuje při nahrávání nového kódu na M4 jádro.

4.1.1 Kompilace kódu pro M4 jádro

NXP pro M4 jádro připravilo SDK nazvané MCUXpresso SDK obsahující jednoduchou vrstvu nad registry periférií a vrstvu pro integraci FreeRTOS.

Připravené ukázky je možné zkompilovat pro M4 jádro pomocí IAR Embedded Workbench nebo nástrojů s prefixem `arm-none-eabi-` a build nástroje CMake. Během práce byla přidána podpora pro MCUXpresso IDE ze strany NXP.

CMake varianta vyžaduje oficiální ARM GCC kompilátor, který není pro architekturu ARM64 předkompilován. Balíček v distribuci má kompilátor zabalený s jinou adresářovou strukturou a proto nelze bez úprav provést kompilaci na minipočítači. CMake konfigurace má kolem tisíce řádků, cesty ke zdrojovým souborům SDK jsou relativní vůči aktuálnímu adresáři a zbytek konfigurace obsahuje duplicitní volby pro `release` a `debug` verzi.

4.1.2 Nahrávání kódu na M4 jádro

Instrukce a data programu musí být uložené v paměti podporující jeho vykonávání. U běžných mikrokontrolerů se instrukce s daty nahrávají do non-volatilní flash paměti, která data uchová i po odpojení napájení. Procesor i.MX 8M obsahuje místo toho 3 typy volatilních pamětí, do kterých lze nahrát instrukce s daty.

První z nich je *Tightly-Coupled Memory* (TCM), která je nejblíže M4 jádru a zaručuje nízké deterministické latence, které jsou nutné pro splnění real-time podmínek. Paměť TCM o velikosti 256 kB je rozdělena na dvě stejně velké částí TCML a TCMH, které mají vlastní cache. TCML se používá pro instrukce programu a TCMH pro jeho paměť.

Další paměti je OCRAM o velikosti 128 kB. Celý program lze také provozovat z DDR paměti, u které nemusí být nízké latence při přístupu, což může například zapříčinit zpožděný výběr instrukcí pro zpracování přerušení. Dále je také nutné vyhradit paměť v Device Tree konfiguraci.

Oficiálně podporované nahrávání kódu je pomocí rozhraní JTAG nebo v zavaděči ještě před samostatným zavedením operačního systému [12]. Metoda nahrávání pomocí JTAG nebude popsána, protože PICO-PI-IMX8M nemá rozhraní vyvedené.

Nahrávání v zavaděči Das U-Boot probíhá nakopírováním programu například z `/boot` oddílu do paměti TCM, která je z A53 jádra dostupná na adrese `0x7e_0000`. Dále proběhne spuštění M4 jádra zápisem do registru `SRC_M4RCR` [13] a zavedení systému Linux pomocí příkazu `boot`. Potřebné příkazy lze nalézt ve výpise 6.

```
u-boot=> fatload mmc 1:1 0x7e0000 hello.bin
u-boot=> bootaux 0x7e0000
u-boot=> boot
```

Výpis 6: Nahrání aplikace na M4 jádro a následné zavedení Linuxu zavaděčem

Takový postup však není efektivní při vývoji, protože vyžaduje opakovat zdlouhavý proces zavádění operačního systému a startování aplikace na obou typech jader. Další nepříjemnou vlastnosti je nutnost kopírovat program do oddílu `/boot` nebo využít protokol TFTP pro stáhnutí nového M4 kódu.

Pro SoC i.MX 6SoloX[22] a 7Dual existují programy[23] pro nahrání nového kódu a ovládání M4 jádra přímo z userspace. Pro ovládání jádra si nahrávací program namapuje registr `SRC_M4RCR` a také TCM paměť do svého adresního prostoru pomocí systémového volání `mmap` nad znakovým zařízením `/dev/mem`. Zařízení obsahuje celou dostupnou paměť z pohledu aplikačního procesoru, ale operace nad zařízením vyžadují běh pod uživatelem root. Nahrání nového kódu pak probíhá pouhým překopírováním instrukcí do namapované paměti a start jádra zápisem do registru `SRC_M4RCR`.

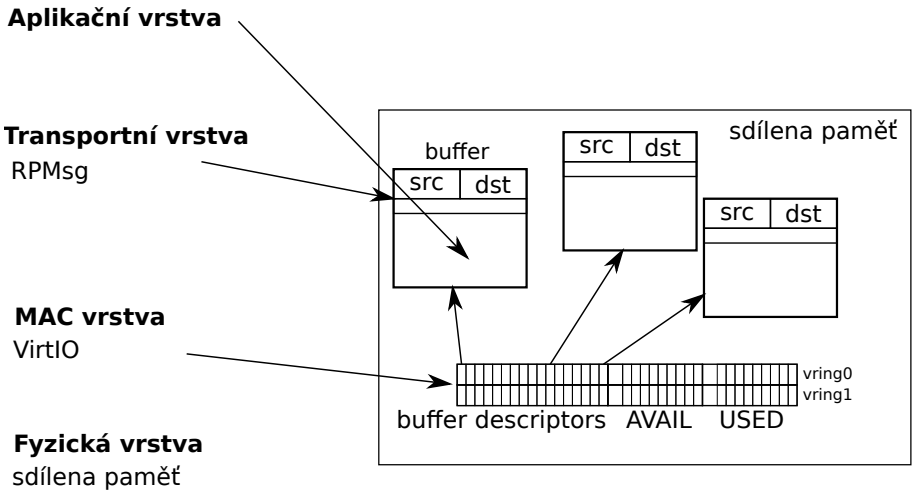
4.1.3 Komunikace mezi jádry

Komunikace mezi jádry probíhá zasíláním bufferů pomocí frameworku RPMsg. Na framework lze nahlížet na obrázku 6 jako na vícevrstvý model podobný modelu TCP/IP [14].

Aplikační vrstva obsahující data aplikace je zapouzdřena v transportní vrstvě RPMsg. Jednotlivé buffery jsou v RPMsg přenášeny kanály jednoznačně identifikovanými zdrojovou a cílovou adresou. V aktuální verzi jádra se využívají pouze dynamicky vytvořené kanály, kdy jedna strana musí speciální zprávou ohlásit textový název kanálu a svojí adresu. V další komunikaci se pak už využívá výhradně číselné adresování. Součástí hlavičky zpráv je také velikost aplikačních dat.

Na vrstvě přístupu k médiu (MAC) je využít jaderný framework VirtIO. Framework byl primárně navrhnut pro zefektivnění diskové a sítové komunikace mezi virtuálním strojem a hostitelem, kdy není potřeba emulovat funkcionalitu hardware [15].

[22] `https://github.com/ektor5/udooneo-m4uploader`
[23] `https://github.com/NXPmicro/imx-m4fwloader`



Obrázek 6: Vícevrstvý model RPMsg

Pro obousměrnou komunikaci jsou ve sdílené paměti uloženy dvě struktury nazvané jako `vring`. První struktura `vring` se využívá pro přenos dat z M4 jádra do Linuxu a druhá pro přenos dat opačným směrem. Každá `vring` struktura se skládá ze tří částí v souvislé paměti. První část *buffer descriptors* je v textu označována jako tabulka deskriptorů, *available buffers* jako kruhový buffer AVAIL a *used buffers* je označován jako kruhový buffer USED. V jednotlivých částech jsou popsány pouze důležité atributy, protože některé nemají využití, či se jedná pouze příznaky. Počet deskriptorů ovlivňuje celkovou velikost struktury `vring` a proto je nutné, aby obě jádra znala počet deskriptorů.

Tabulka deskriptorů je pole struktur složené z adresy bufferu ve sdílené paměti a její velikosti, která je v aktuální implementaci konstantní. Další dvě části obsahují kruhový buffer sloužící pro předávání deskriptorů. Oproti klasickému kruhovému bufferu, obsahujícímu indexy hlavy a ocasu, je zde pouze index ocasu. Index hlavy je uložen v lokální paměti daného jádra. Tato změna je způsobena využitím principu *single-reader single-writer*, kdy pouze jedno jádro může zapisovat do části paměti. Díky této úpravě se jedná o *lock-free* strukturu, která nevyžaduje žádnou synchronizaci [14], která by v tomhle případě vyžadovala podporu ze strany hardware. Předejde se tak hůře laditelným synchronizačním problémům a možné degradaci výkonu.

Na nejnižší vrstvě je sdílená paměť a nějaká forma notifikací. Sdílená paměť o velikosti 1 MB, začínající na adrese `0xb800_0000`, je rezervována v Device Tree konfiguraci[24], aby ji Linux nealokoval pro jiné účely. V této paměti jsou pouze dvě `vring` struktury. Buffery deskriptorů jsou alokovány z globálního memory pool.

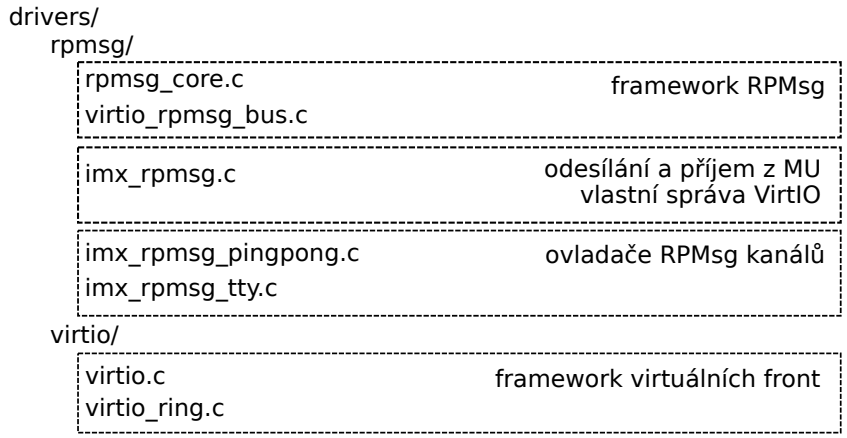
Pro zasílání notifikaci mezi jádry se využívá hardwarový modul *Messaging Unit* (MU), který je rozdělen na komunikující stranu A a B. Na každé straně jsou k dispozici 4 přerušení, která může jedna strana využít pro notifikovaní druhé strany a tím zpřístupnit meziprocesorovou

[24] `https://source.codeaurora.org/external/imx/linux-imx/tree/arch/arm64/boot/dts/freescale/fsl-imx8mq.dtsi?id=5e23f9d6114784d77fd4ed5848953356c3575532#n84`

komunikaci. Dále každá strana obsahuje čtyři 32bitové registry, které lze nastavit tak, aby při jejích zápisu či čtení byla druhá strana přerušena. Tyto registry se také nazývají *mailbox*. Po nastavení přerušení pro každý mailbox, lze zároveň s notifikací přenášet i 32bitovou zprávu. Pokud se nastaví přerušení jenom pro poslední mailbox, tak je možné atomicky přenášet zprávy o délce 4×32 bitů. Vyžaduje to však, aby vysílající strana nezapsala do těchto registrů dřív, než si je druhá strana vyzvedne. Informaci, zda registry byly přijaty druhou stranou, lze zjistit kontrolou `TEx` ve status registru MU, nebo pomoci přerušení.

Aktuální implementace RPMsg využívá pouze jeden 32bitový mailbox, ve kterém se přenáší informace ve které `vring` struktuře došlo ke změně. Přehled jaderných modulů potřebných pro RPMsg lze vidět na obrázku 7.

Pro zaslání bufferu z M4 jádra do Linuxu musí M4 jádro počkat na buffer v cyklickém bufferu AVAIL. Cyklický buffer je prázdný, pokud je lokálně uložená hlava shodná s ocasem, uloženým ve sdílené paměti. Při neplatnosti podmínky se vyzvedne index na buffer deskriptor a inkrementuje lokální index hlavy. Následně proběhne dereference adresy bufferu a vloží se na začátek RPMsg hlavička, následována aplikačními daty. Následně je index buffer deskriptoru uložen do cyklické fronty USED a inkrementován ocas ve sdílené paměti. Na závěr proběhne notifikace Linuxu o změně v cyklickém bufferu. Jednoduchou implementaci na straně M4 jádra lze nalézt v ukázce `m4/rpmsg`[25].



Obrázek 7: RPMsg ve zdrojových kódech jádra Linuxu

4.1.4 Problém aktuálního řešení při vývoji

Při restartu M4 aplikace dochází ke ztrátě lokálně uložených indexů kruhových bufferů. Po restartu má M4 aplikace nastavený index hlavy na první prvek a kvůli tomu dojde k příjmu již dříve přijatých zpráv. Problém lze například řešit značením zpracovaných dat v aplikačních datech [4]. I kdyby se tato data ukládala do perzistentní paměti, tak by zde mohly vznikat

[25] `https://github.com/trnila/picopi8m-ros-demos/tree/master/m4/rpmsg`

situace vedoucí k nutnému restartu celého minipočítače. Situaci může být například nekorektní aplikace, která chybou přepíše `vring` struktury.

Dalším problémem může být únik (leak) bufferů, kdy je aplikace z důvodu chyby nevrátí zpět. Pokud dojde k únikům všech bufferů, tak nebude možné například odeslat žádnou zprávu z Linuxu a bude nutné provést restart celého minipočítače. Tento problém se může častěji vyskytovat u varianty *zero copy*, protože aplikace vrací buffer až po zpracování.

Dalším problémem jsou zaregistrované RPMsg kanály, které by se musely například odregistrovat před startem aplikace.

Z těchto důvodů by bylo vhodné při restartu M4 jádra `vring` struktury reinicializovat, aby M4 jádro mohlo přistupovat ke strukturám, které jsou v konzistentním stavu. Dále by bylo vhodné, aby operační systém prováděl ovládání a nahrávání kódu M4 jádra a nebyl nutný userspace program přistupující do fyzické paměti.

4.2 Restartování jaderného modulu `imx_rpmsg`

Úpravy monolitického jádra jsou poněkud složitější a mohou vést k nestabilitě, protože všechny moduly mají přístup k celé paměti a mohou se tak navzájem ovlivňovat. Proto první jednoduchou cestou jak zajistit reinicializovaní struktury `vring` je odebrání a následné vložení jaderného modulu `imx_rpmsg`. V aktuální implementaci je daný modul zkompilován přímo do jádra, a proto neobsahoval funkce pro uvolnění prostředků. Nejprve je nutné v souboru `KBuild` nastavit, aby se daný modul překládal jako dynamický modul. Následně je nutné při odebírání modulu uvolnit alokované zdroje včetně VirtIO zařízení.[26]

Během testování se však vyskytl při nahrávání kódu další problém. Dokud si M4 aplikace nevyzvedne zprávu z MU registru, dochází k aktivnímu čekání v kódu linuxového jádra do doby, než bude zpráva přijata. Problém muže nastat při zastavení M4 procesoru bez jeho odhlášení z RPMsg kanálu nebo když M4 aplikace nepřečte zprávu z MU registru. Kdykoliv pak jaderný ovladač zapíše další zprávu do kanálu, dojde k zacyklení jednoho A53 jádra do restartu systému. Aktuálním jednoduchým řešením je čekat pouze nějaký konečný čas a poté nahlásit chybu.[27]

Po těchto úpravách je možné reinicializovat `vring` struktury pomocí příkazu `rmmod imx_rpmsg && modprobe imx_rpmsg` během zastaveného M4 jádra. Nejedná se však o spolehlivé řešení, protože není zajištěno, aby bylo M4 jádro zastavené.

4.3 Nahrávání kódu pomocí jaderného frameworku remoteproc

Jaderný framework remoteproc sjednocuje startování, vypínání a nahrávání kódu na další jádra procesoru [16]. Framework ale není ze strany NXP podporován. Ovládání jádra je možné provést v userspace pomocí zápisu do souboru v debugfs, či nověji v sysfs. Podstatným rozdílem je také to, že se využívá program ve formátu *elf*, ze kterého se potřebné sekce nakopírují do patřičných

[26] `https://github.com/trnila/linux-tn/commit/4f77d7492635a62ee71ade6c0a95eb0e43501b27`
[27] `https://github.com/trnila/linux-tn/commit/13c7b0310167d71c53ae3ab203960c30ef98aac2`

míst definovaných linker skriptem. Dalším rozdílem je existence sekce `resource_table`, ve které je možné definovat zdroje, jako je RPMsg VirtIO zařízení. Po zastavení jádra jsou zdroje uvolněny a další start M4 jádra reinicializuje `vring` struktury.

4.3.1 Ovládání a nahrávání kódu

V novější verzi linuxového jádra byl začleněn ovladač `imx_rproc`, který provádí ovládání jader na procesorech ze série i.MX 6 a i.MX 7. Ovladač je v rámci práce do linuxového jádra začleněn a doplněn paměťovou mapou definující adresu TCM paměti z pohledu M4 jádra a aplikačního jádra.

Během nahrávání kódu do TCM paměti docházelo k pádům linuxového jádra z důvodu kopírování pomocí funkce `memcpy`. TCM paměť je do jádra mapována pomocí funkce `ioremap` a vyžaduje, aby paměťové operace byly zarovnány. Proto je nutné při nahrávání kódu použít funkci `memcpy_toio` v `drivers/remoteproc/remoteproc_elf_loader.c`. Dalším problémem bylo, že druhé nahrání kódu do TCM paměti se neprojevilo z důvodu odstraněného bitu `ENABLE_M4` v registru `SRC_M4RCR` ovladačem `imx_rproc.c`.

Ovladač se pro M4 jádro zavede v případě existence uzlu s danou hodnotou vlastnosti `compatible` v Device Tree konfiguraci. Pro i.MX 8M je připravena hodnota `fsl,imx8m-cm4` a ukázkový uzel lze vidět ve výpise 7. V případě existence více M4 jader by jen stačilo přidat více takových uzlů s případnou identifikací M4 jádra atributem `reg`, kterou by ovladač mohl použít jako posunutí v adresách registrů.

```
imx8m-cm4 {
   compatible = "fsl,imx8m-cm4";
   syscon = <&src>;
};
```

Výpis 7: Device Tree uzel pro M4 jádro využívající ovladač `imx_rproc`

Po těchto úpravách je možné nahrávat nový kód ve formátu *elf* a ovládat M4 jádro pomocí rozhraní debugfs. Kód M4 jádra je hledán pod určitým názvem v několika cestách, jako je například `/lib/firmware`. Ukázku, jak nahrát kód na M4 jádro a následně spustit, lze vidět ve výpise 8. V této fázi již není potřebný externí userspace program, přistupující do TCM paměti či do paměti registrů.

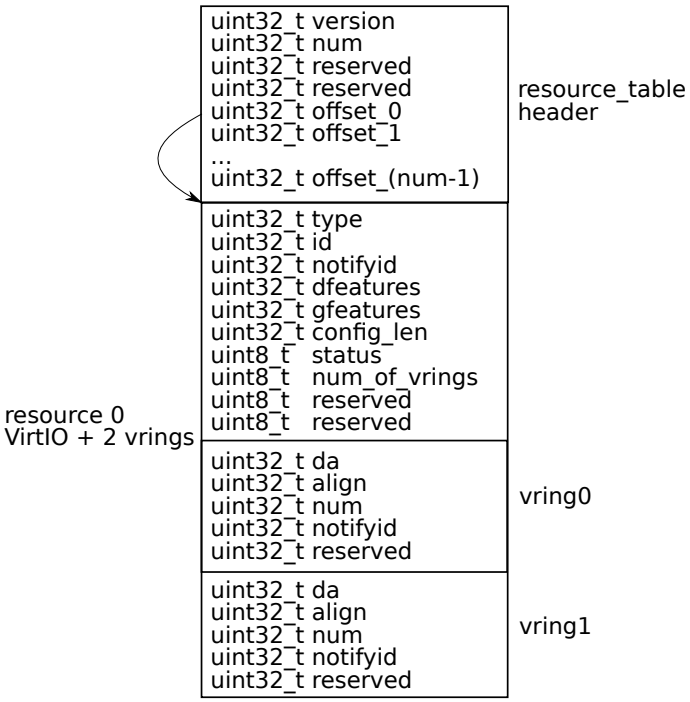
```
$ echo stop > /sys/kernel/debug/remoteproc/remoteproc0/state
$ cp ./build/debug/hello /lib/firmware/rproc-imx-rproc-fw
$ echo start > /sys/kernel/debug/remoteproc/remoteproc0/state
```

Výpis 8: Nahrání aplikace na M4 jádro pomocí `remoteproc`

4.3.2 Tabulka zdrojů `resource_table`

Tabulka zdrojů definuje potřebné zdroje M4 jádra. Tyto zdroje jsou před startem jádra alokovány a při zastavení dealokovány. Nejčastějším zdrojem je VirtIO, který se využívá pro komunikaci mezi M4 jádrem a Linuxem. Dále lze například alokovat paměť pro M4 jádro nebo také využít zdroj `TRACE`, který zpřístupní paměť do userspace pomocí debugfs. Lze tak například do Linuxu poskytnout záznam či aktuální nastavení proměnných.

Při alokaci remoteproc zařízení se přiřazuje výchozí *elf loader*, který načítá sekce programu do paměti. Tyto sekce jsou definovány v linker skriptu během kompilace. Výchozí *elf loader* ze sekce `resource_table` poskytne konfiguraci všech zdrojů, které se následně alokují a nakonfigurují. Sekce je složena z hlavičky a zdrojů v souvislé paměti. Libovolné zdroje potřebují různý počet parametrů a proto je součástí hlavičky, jak je vidět na obrázku 8, i offset na každý zdroj.



Obrázek 8: Sekce `resource_table` s jedním zdrojem VirtIO

Sekci `resource_table` lze definovat v jazyce C/C++ pomocí rozšířené syntaxe kompilátoru `gcc`. K proměnným, které se mají vložit do sekce, se přidá atribut `section` s názvem sekce. Ukázku, jak vytvořit prázdnou tabulku lze vidět ve výpise 9. Pro složitější tabulky je vhodnější použít struktury. Nutno podotknout, že sekce musí být inicializována v době překladu.

Tabulka je povinná, i když není potřeba žádný zdroj. Aby nebylo nutné tabulku definovat v programech, které žádné zdroje nepotřebují, byl upraven linker skript, aby vždy do sekce vložil případnou existující sekci a za ní vložil hlavičku prázdné `resource_table` tabulky. V případě, že programátor nedefinuje tabulku, bude na začátku právě tabulka bez zdrojů. Potřebnou úpravu linker skriptu lze vidět ve výpise 10. Celá sekce se zarovnává na násobek 256. Pro ověření lze

```
__attribute__((section(".resource_table"))) uint32_t resources[] = {
  1 /*version*/, 0 /*num of resources*/, 0 /*reserved*/, 0 /*reserved*/
};
```

Výpis 9: Ukázka vytvoření `resource_table` sekce bez zdrojů

obsah sekce extrahovat z *elf* pomocí programu `objdump`. Ve výpise 11 je obsah sekce se zdrojem VirtIO pro RPMsg. Data jsou zde uložena ve formátu little-endian. Prvních 32 bitů definuje první verzi tabulky zdrojů.

```
.resource_table : {
   *(.resource_table) /* place user .resource_table */
   /* place default resource table if no .resource_table provided from user */
   LONG(1) /* version 1 */
   LONG(0) /* 0 resources */
   . = ALIGN(0x100);
}
```

Výpis 10: Linker skript pro výchozí tabulku bez zdrojů

```
[root@picopi m4]# objdump -s -j .resource_table build/debug/linecam_raw
build/debug/linecam_raw:  file format elf32-littlearm
Contents of section .resource_table:
20000000 01000000 01000000 00000000 00000000 ................
20000010 14000000 03000000 07000000 00000000 ................
20000020 01000000 00000000 00000000 00020000 ................
20000030 ffffffff 00100000 00010000 01000000 ................
20000040 00000000 ffffffff 00100000 00010000 ................
20000050 02000000 00000000 01000000 00000000 ................
20000060 00000000 00000000 00000000 00000000 ................
```

Výpis 11: Binární obsah `resource_table` sekce

4.3.3 Zdroj VirtIO pro RPMsg komunikaci

Pro alokaci VirtIO zařízení je potřeba vytvořit tabulku `resource_table` dle obrázku 8. Po následném startu se vytvoří VirtIO zařízení dle definovaných parametrů. V této fázi není stále možná efektivní RPMsg komunikace, protože druhá strana se nedozví o existenci nové zprávy. Ovladač `imx_rproc` musí dále implementovat funkci pro notifikaci M4 jádra a také přijímat notifikace z M4 jádra o zaslání nebo navrácení bufferu. Pro notifikace se používá Messaging Unit (MU), který při zápisu do mailboxu vyvolá přerušení na druhé straně.

Funkcionalita je implementována v ovladači `imx_rpmsg.c`, ale nelze ji využít, protože si spravuje své vlastní VirtIO zařízení. Z tohoto důvodu musí být funkcionalita implementována přímo v ovladači `imx_rproc`. MU je definována v Device Tree uzlu `mu@30aa0000`. Tento uzel ovladač využije k namapování MU registrů do paměti a inicializaci hodin či přerušení.

Pro notifikaci M4 jádra je nutné implementovat remoteproc funkcionalitu `kick`, ve které se do MU mailboxu vloží číslo `vring`, kde došlo ke změně. Po zápisu je vyvoláno na M4 jádře přerušení, které obsluhuje IRQ handler `MU_M4_IRQHandler`. M4 jádro může následně přijmout buffery od svého lokální indexu hlavy do ocasu v cyklickém bufferu AVAIL.

Příjem notifikací na straně Linuxu je o něco složitější. Nejprve je potřeba povolit pro konkrétní mailbox přerušení a zaregistrovat obslužnou funkci, která zavolá `rproc_vq_interrupt` s informaci, na které `vring` struktuře došlo ke změně. Funkce nesmí být volaná z přerušení, ale je možné ji obsloužit v jaderném vlákně. Pro takový případ existuje koncept *threaded IRQ*, kdy se pomocí `request_threaded_irq` zaregistrují dvě funkce pro obsluhu přerušení. První funkce obslouží přerušení přečtením zprávy, aby nedošlo k opětovnému vyvolání, a vrátí informaci pro dokončení obsluhy v druhé funkci, běžící již v připraveném jaderném vlákně. Obsluhující vlákno lze také vidět v tabulce procesů - jeho název může být například `[irq/46-imx-rpro]`.

Pro zaregistrování obslužných funkcí byla použita funkce `devm_request_threaded_irq`. Funkce s prefixem `devm_` ukládají všechny alokované zdroje k zařízení. Při odebrání zařízení z ovladače jsou tyto zdroje automaticky dealokovány. Princip zjednodušuje jaderný kód a zamezuje chybám, kdy není zdroj v případě chyby dealokován.

Remoteproc `imx_proc` ovladač je v této fázi již plně funkční pro RPMsg komunikaci. Zaslání zprávy či vráceni bufferu z M4 jádra pak probíhá modifikaci `vring` struktury a následným zápisem čísla modifikované `vring` struktury do mailboxu. Zápis vede k vyvolání přerušení na aplikačním jádře, kde se nejprve provede obsluha přerušení a následně se notifikace dokončí v již zmiňovaném vlákně.

4.3.4 Alokace `vring` společně s buffery v omezené paměti

Nutno podotknout, že v současné době není jádro schopné alokovat `vring` struktury na adrese definované v tabulce zdrojů. Alokace struktury `vring` společně s buffery se provádí pomocí funkce `dma_alloc_coherent`, která alokuje koherentní paměť z rezervované oblasti použité i u dalších ovladačů. Adresa alokované `vring` se následně uloží zpět do sekce `resource_table` odkud ji může knihovna `rpmsg_lite` využít. Protože se při každém restartu provádí alokace VirtIO znovu, tak může docházet k alokaci `vring` struktur na jiných místech v paměti, což může být problematické při nastavování ochrany paměti na M4 jádře.

Funkce `dma_alloc_coherent` alokuje paměť z oblasti přiřazené v zařízení ovladače `imx_rproc`. Oblast lze k Device Tree uzlu `imx8m-cm4` přiřadit pomocí atributu `memory-region`. Ten se musí odkazovat na rezervovanou paměť definovanou jako `shared-dma-pool`. Následně je nutné v ovladači oblast k zařízení přidat pomocí funkce `of_reserved_mem_device_init`. Po této úpravě je jádro schopno deterministicky alokovat obě `vring` struktury společně s buffery

v menší paměťové oblasti, kterou lze jednoduše povolit v ochraně paměti, která je popsána v kapitole 5.2. Výslednou Device Tree konfigurací lze vidět ve výpise 12.

```
/ {
   reserved-memory {
      ranges;
      #address-cells = <0x2>;
      #size-cells = <0x2>;

      rpmsg_reserved: rpmsg@0xb8000000 {
         compatible = "shared-dma-pool";
         no-map;
         reg = <0x0 0xb8000000 0x0 0x100000>;
      };
   };

   imx8m-cm4 {
      compatible = "fsl,imx8m-cm4";
      memory-region = <&rpmsg_reserved>;
      syscon = <&src>;
   };
};
```

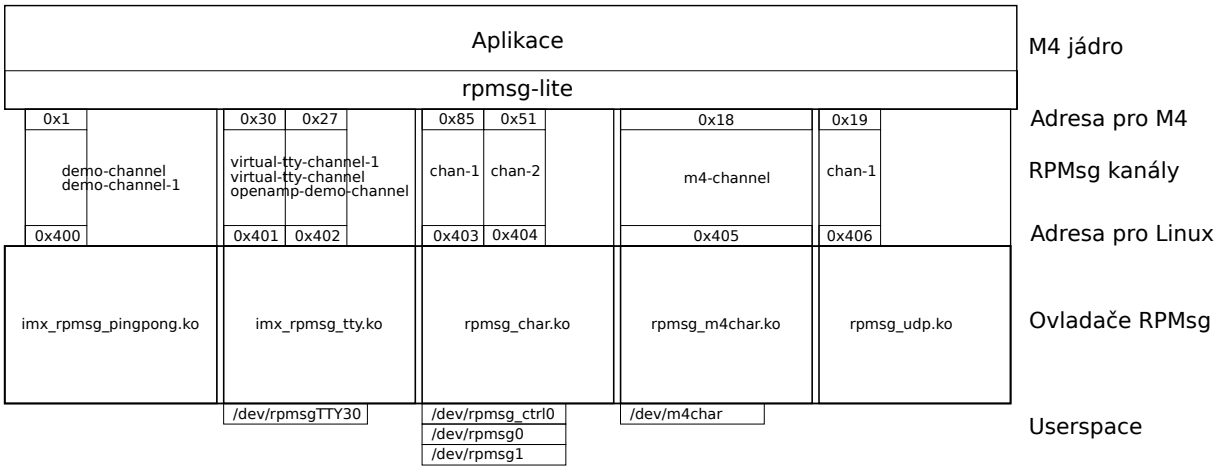
Výpis 12: Výsledná Device Tree konfigurace pro M4 jádro i s rezervovanou pamětí

4.4 RPMsg ovladače

Komunikaci v rámci kanálu zajišťují RPMsg ovladače jejichž schéma lze vidět na obrázku 9. V případě, že M4 jádro pošle speciální zprávu `NS_ANNOUNCEMENT` s daným názvem kanálu, dojde k vyvolání funkce `probe` ovladače. Při zániku kanálu, který se mimo jiné provádí i při zastavení jádra, dojde k vyvolání funkce `remove`. Další funkce `callback` je vyvolána při příjmu nové zprávy na kanálu. Ovladač může kdykoliv poslat zprávu pomocí funkce `rpmsg_send`. Musí si však pohlídat, aby během zasílání nebyl RPMsg odstraněn.

4.5 Problém současných RPMsg ovladačů

NXP dodává dva ukázkové ovladače. První ovladač `imx_rpmsg_pingpong` při příjmu zvýší číslo o jedna a pošle zpět v případě, že je menší než pevně nastavený limit. Druhý ovladač `imx_rpmsg_tty` při ohlášení kanálu vytvoří TTY zařízení nazvané `/dev/ttyRPMSGxx`, kde `xx` je číslo adresy na straně M4. Zaslané a přijaté zprávy jsou sjednoceny do jednoho proudu dat, který je nutné parsovat. Ovladač ale předpokládá, že nedojde k odstranění kanálu během toho, kdy je deskriptor otevřený v userspace programu. Při této situaci systémové volání `write` přistoupí na neplatný ukazatel RPMsg a dojde k pádu linuxového jádra.



Obrázek 9: RPMsg ovladače a jejich návaznost na userspace

V hlavní větvi linuxového jádra existuje ovladač `rpmsg_char`, umožňující dynamicky vytvářet endpoint pro kanály libovolného názvu pomocí řídícího znakového zařízení `/dev/rpmsg_ctrl0`. Systémové volání `ioctl` pomocí požadavku `RPMSG_CREATE_EPT_IOCTL` a předané struktury vytvoří nový deskriptor pro požadovaný kanál. Systémová volání `read` a `write` pracují s jednotlivými RPMsg buffery a lze tak atomicky přenášet až 512bajtové zprávy bez nutnosti hledat začátek další zprávy. Při pokusech s restartováním jádra však také docházelo k souběhu. Pokud se během vytváření nového kanálu odstranilo RPMsg zařízení, tak se podařilo vytvořit znakové zařízení pro neexistující kanál s neplatným ukazatelem na RPMsg, což opět vedlo k chybě.

4.5.1 Ovladač `rpmsg_m4char`

V rámci práce je naimplementovaný zjednodušený ovladač `rpmsg_m4char`, který byl inspirován výše zmíněným ovladačem `rpmsg_char`. Ovladač po nahrání zpřístupní znakový soubor `/dev/m4char` po celou dobu běhu systému a není nutné řešit jeho dynamickou tvorbu při dočasném zániku kanálu během restartu M4 jádra. Dále byl touto úpravou odstraněn problém se souběhem při vytváření nového kanálu a odstraněním RPMsg zařízení. Pro atomické zasílání zpráv stačí jen otevřít soubor a provést systémové volání `read` či `write`. Není tedy potřeba aplikačně řešit oddělování jednotlivých zpráv a je tak možné komunikovat pomocí `bash` skriptu.

V případě neohlášeného kanálu `m4-channel` ovladač pro volání `read` a `write` vrací ihned chybu `EPIPE`. Při ohlášeném kanálu a dostupných RPMsg bufferech systémové volání `write` zasílá buffer pomocí funkce `rpmsg_send`.

Čtení nad zařízením vrací jeden celý buffer z fronty. V případě, že žádný další buffer není k dispozici, tak je uživatelský proces pomocí funkce `wait_event_interruptible` uspán do doby, než přijde nový RPMsg buffer. Přijatý buffer z M4 jádra je ve funkci `callback` překopírován do nově alokované paměti a zařazen na konec fronty. Následně je probuzen uspaný proces pomocí funkce `wake_up_interruptible`, ve kterém se provede odebrání prvního bufferu z fronty a na-

kopírování do userspace. Suffix funkce `_interruptible` zajistí, že proces půjde přerušit pomocí signálu. V opačném případě by nebylo možné daný proces ani ukončit do příjmu bufferu.

Při odstranění RPMsg zařízení nebo kanálu, dojde k zavolání funkce `remove`. Při příjmu bufferů není potřeba řešit odebrané zařízení, protože funkci `callback` volá RPMsg framework. Během odesílání bufferů je však nutné, aby zařízení nebylo odebráno. Z tohoto důvodu je vytvořen mutex, který je po celou dobu odesílání zamknutý, což vede k následujícímu problému. Při odebrání zařízení je funkce `remove` blokována, dokud se nedokončí přenos dat, aby se mohl daný mutex zamknout a nastavit nulový ukazatel RPMsg. V případě nedostatku volných RPMsg bufferů funkce `rpmsg_send` daný proces uspí na maximální čas 15 sekund, či do vrácení bufferu ze strany M4 jádra. Během tohoto stavu je mutex zamknutý a nelze restartovat M4 jádro, což vede k problému, kdy není ani možné zastavit jádro a nahrát nový kód. Pro daný problém nebylo nalezeno žádné řešení, protože zde chybí zpětná vazba o přijetí volného bufferu. Z tohoto důvodu je systémové volání `write` automaticky neblokující.

4.5.2 Ovladač `rpmsg_udp`

Součástí ukázek je také naimplementovaný ovladač `rpmsg_udp`[28], který přijatá data z M4 jádra posílá ihned po UDP na nastavenou cílovou adresu. Adresa může být IPv6 nebo IPv4 a lze posílat i na multicast či broadcast adresu. Pomocí ovladače lze velice jednoduše zasílat data z M4 jádra bez nutnosti implementace UDP stacku či běhu userspace aplikace.

4.6 Kompilace kódu pro M4 jádro

V kapitole 4.1.1 je zmíněna složitost kompilace a z tohoto důvodu je provedena úprava vhodná pro zjednodušení dále vytvořených ukázek. Pro nástroj CMake je vytvořena čtveřice CMake knihoven[29] - `m4sdk`, `freertos`, `rpmsg` a `rosserial_rpmsg`. Konfigurace se díky tomu zjednodušila na pár přehledných řádků a navíc je možné v jednom adresáři zkompilovat více programů s odlišnými knihovnami. Různá nastavení v souborech `clock_config.c` a `board.c` jsou pro všechny ukázky stejné a proto se během sestavení kompilují automaticky. Výsledný `elf` soubor je slinkován pomocí upraveného linker skriptu `MIMX8MQ6xxxJZ_cm4_ram.ld`, který přidává výchozí prázdnou sekci `resource_table`. Ukázku zjednodušené konfigurace lze vidět ve výpise 13.

Dále je upraven CMake toolchain, aby nevyžadoval nástroje `arm-none-eabi-` v předem definovaném adresáři, ale hledal je v proměnné prostředí `PATH`. Díky této úpravě je možné využít cross kompilátor `arm-none-eabi-gcc` z distribuce a mít možnost překládat M4 aplikace přímo na minipočítači. V době psaní této práce nebylo možné využít balíček z distribuce z důvodu absence knihoven s podporou pro hard float. Problém je ve vytvořené distribuci vyřešen instalací novějších balíčků `libnewlib-dev` a `libnewlib-arm-none-eabi`.

[28] `https://github.com/trnila/picopi8m-ros-demos/tree/master/rpmsg/udp`

[29] `https://github.com/trnila/picopi-m4sdk/tree/freertos_SDK_2.3_IMX8M/tools/cmake_toolchain_files`

```
cmake_minimum_required(VERSION 3.0.0)
project(cmake_usage)
find_package(M4SDK REQUIRED)
find_package(FreeRTOS REQUIRED)
find_package(rpmsg REQUIRED)
include_directories(.)

add_firmware(bare bare.c pin_mux.c)
target_link_libraries(bare mcuxpressosdk)

add_firmware(rtos rtos.c pin_mux.c)
target_link_libraries(rtos mcuxpressosdk freertos)

add_firmware(rpmsg rpmsg.c pin_mux.c)
target_link_libraries(rpmsg mcuxpressosdk freertos rpmsg_lite)
```

Výpis 13: Ukázka přehlednější CMake konfigurace pro kompilaci M4 aplikací

Pro jednodušší práci je vytvořen nástroj `m4build`, který provádí kompilaci M4 aplikací dle souboru `CMakeLists.txt` v aktuálním adresáři. Nástroj nastaví cestu k CMake toolchain a dle potřeby vytvoří adresář `build`, ve kterém z CMake konfigurace vygeneruje `Makefile` soubor. Následně se provede kompilace. Zkompilovanou aplikaci lze pak najít v adresáři `build/debug/` jak ve formátu *elf* obsahující ladicí symboly, tak v binárním formátu, který je procesor schopný rovnou vykonávat.

Pro pohodlné nahrávání je vytvořena aplikace `m4ctl`, která umí procesor zastavit, spustit nebo také nahrát instrukce ze souboru do paměti - jak lze vidět ve výpise 14. Výhodnější je však využít nástroj `m4run` v adresáři s M4 aplikací, který před nahráním provede kompilaci pomocí již zmíněného nástroje `m4build`.

```
$ m4build
$ m4run example1
$ m4ctl start ./build/debug/hello
$ m4ctl start
$ m4ctl stop
```

Výpis 14: Ukázka funkcionality programů pro kompilaci či nahrávání kódu do M4 jádra

5 Využití periférií

V současném stavu systém Linux předpokládá, že jako jediný přistupuje k periferiím a tím nemůže vznikat souběh, který by například zapříčinil ztrátu nastavení.

M4 jádro má také přístup do prvních dvou gigabajtů DDR paměti, čímž vznikají nové problémy a možnosti napadení systému. Chyba v programu může vést k modifikaci paměti linuxového jádra, linuxových aplikací nebo také bufferů. Může se tak stát, že se při zápisu dat z linuxové aplikace zapíší jiná data na úložiště, protože mezi tím byl modifikován buffer z M4 jádra. V rámci práce je demonstrována jednoduchá ukázka, jak docílit eskalaci oprávnění v běžícím linuxovém procesu. Tento problém může vytvářet chyby, které mohou způsobovat nedeterministické chování, a proto bude řešení vysvětleno v jedné z následujících podkapitol.

V kapitole budou následně popsány dostupné periférie minipočítače a jak je použít na M4 jádře, ale také v systému Linux. Vytvořené ukázky pro samostatné periférie jsou v adresáři `m4` a `linux_periph` repositáře `picopi8m-ros-demos`.[30]

5.1 Souběh při sdílení periférií

Registry periferií jsou namapované do adresního prostoru jader. Na architektuře ARM je možné přistupovat do paměti pomocí instrukcí `ldr` (load) a `str` (store).

Ve výpise 15 lze vidět potřebné instrukce pro nastavení logické úrovně na pinu `GPIO4_IO26`. Reference na čísla řádků v tomto odstavci odkazují na tentýž výpis. Adresu `0x3023_0000` registru periférie `GPIO4` nelze zakódovat do instrukce `ldr`. Z tohoto důvodu instrukce na řádku č. 3 načítá adresu periférie z paměti na adrese `pc + 24` (řádek č. 7) do registru `r2`, kde registr `pc` obsahuje adresu aktuální instrukce. Další instrukce `ldr` na řádku č. 4 provádí dereferenci a tím se uloží aktuální nastavení hodnot pinů do registru `r3`. Následně probíhá na řádku č. 5 logická operace OR nad aktuálním nastavením pinů a hodnotou `0x400_0000`, která představuje jedničku na 26. bitu. Výsledek operace je uložen do registru `r3` a následně na řádku č. 6 uložen zpátky do paměti periférie.

Při nastavování pinů na M4 jádře a Linuxu dojde k souběhu, který povede k zahození změn druhého jádra, protože první jádro přečetlo stav paměti dřív, než druhé jádro uložilo své nastavení do paměti. Demonstraci vzniku souběhu lze vidět v ukázce `gpio-concurency`.[31] V ukázce se na M4 jádře ve smyčce přepíná pin `GPIO4_IO23`. Linux ve smyčce nastavuje pin `GPIO4_IO26` a poté čte jeho hodnotu. V případě, že se neshoduje nastavená hodnota pinu s přečtenou, tak právě došlo k souběhu.

Takové části kódu je nutné umístit do kritických sekcí, které se musí provést atomicky na jednom jádře. Kritické sekce se realizují pomocí synchronizačních primitiv, jako je mutex či jeho varianta s aktivním čekáním nazvaná spinlock. Primitiva lze realizovat pomocí instrukcí `ldrex` a `strex`, které jsou exkluzivní variantou instrukcí `ldr` a `str`.

[30] `https://github.com/trnila/picopi8m-ros-demos`
[31] `https://github.com/trnila/picopi8m-ros-demos/tree/master/gpio-concurency`

```
1. 1ffe70e8 <main>:

2.      *(volatile uint32_t*) 0x30230000 |= 1U << 26;
3. 1ffe711e: 4a06      ldr r2, [pc, #24] ; (1ffe7138 <main+0x50>)
4. 1ffe7120: 6813      ldr r3, [r2, #0]
5. 1ffe7122: f043 6380 orr.w r3, r3, #67108864 ; 0x4000000
6. 1ffe7126: 6013      str r3, [r2, #0]

7. 1ffe7138: 30230000 .word 0x30230000
```

Výpis 15: Instrukce pro nastavení logické jedničky na pinu `GPIO4_IO26`

Ovladače operačního systému Linux a také M4 aplikace běžně zamezují vznikům souběhů, ale předpokládají, že se k ním přistupuje pouze pomocí jejích rozhraní. Není zde podpora pro kritickou sekci mezi Linuxem a M4 jádrem. Řešení tohoto problému se nabízí hned několik a budou postupně vysvětleny podle jejich náročnosti implementace.

Prvním nejjednodušším řešením je využít pštrosí algoritmus, který ignoruje potencionální problémy, protože se vyskytují pouze ojediněle a nemají fatální důsledky. I když se takové řešení zdá být naivní, lze ho výhodně využít, pokud se nastaví na obou typu jader vstupní pin a následně se jen čte jeho logická úroveň. K souběhu může dojít pouze při inicializaci v případě, že se provádí konfigurace ve stejný okamžik.

Dalším řešením je celou periférii vyhradit pouze pro jedno jádro - Device Tree konfigurací, obyčejným nevyužitím nebo pomocí hardwarové podpory Resource Domain Controller (RDC). Obyčejné nevyužití může být problematické, protože si ovladač povoluje přerušení, ve kterém například zruší příznak události nebo zakáže vyvolávání dalších přerušení.

Na straně Linuxu lze periférii zakázat pomocí Device Tree konfigurace. Díky tomu ji ovladač nehledá a nebude využívat. Zakázat lze ale celý GPIO port obsahující všech 32 pinů a to pomocí nastavení vlastnosti `status` u konkrétního uzlu v Device Tree konfiguraci, jak lze vidět ve výpise 16. Ve výpise se překrývá nastavení již existujícího uzlu, který je definován v konfiguraci pro SoC.

```
&gpio3 {
  status = "disabled";
};
```

Výpis 16: Zakázání periférie `GPIO3` v Device Tree konfiguraci

Dalším způsobem je odebrat zařízení zápisem názvu zařízení do souboru `unbind` daného ovladače. Zápisem do souboru `bind` lze zpátky přiřadit zařízení k ovladači. Některé ovladače ale

mohou předpokládat, že k takové změně nedojde a nemají implementované či otestované jejich odebráni za běhu.

Příkazem `echo 30220000.gpio > /sys/bus/platform/drivers/gpio-mxc/unbind` lze odebrat `GPIO3` z ovladače. Odebírají se však i piny, které se právě využívají jiným ovladačem či userspace aplikací. Následný přístup k periférii vede k chybě jádra. Tento problém je způsoben tím, že ovladače nepočítají s možnosti odebrání GPIO portu za běhu systému.

Pomocí RDC lze periférii vyhradit pouze pro jedno jádro. Následně nepovolený přístup je zamítnut vyvoláním přerušení, které není systémem Linux obslouženo a dojde k fatálnímu zastavení systému. Ve výpise 17 je Linuxu odebrán kompletní přístup k periférii GPIO.

Dalším možným řešením je synchronizovaný přístup k periférii pomocí hardwarového semaforu. Je zde však nutná podpora jak v kódu M4 tak i na straně ovladače v Linuxu. Synchronizace je realizovaná kritickou sekci, kdy ovladač před přístupem k registrům zapíše své číslo do semaforu a následně přečte jeho hodnotu. Přístup do kritické sekce je udělen, pokud se přečtená hodnota shoduje se zapsanou. V druhém případě je nutné pokus opakovat. Po ukončení sekce je nutné daný semafor uvolnit. V aktuální době není podpora hardwarového semaforu na straně Linuxu. Zavaděč Das U-Boot má podporu alespoň pro GPIO periférii.[32]

Pro každou aplikaci je nutno zvážit, která varianta je opravdu potřebná, protože i když varianta se semaforem vypadá jako ideální řešení, může dojít k chybě ovladače, který danou periférii po použití neuvolní a tím dojde ke kompletnímu zablokování ovladače či v horším případě celého systému. Použití semaforu také přináší režii, která se může promítnout do časové odezvy.

5.2 Ochrana paměti

M4 jádro má přístup do prvních 2 GB DDR paměti, kde je také paměť Linuxového jádra a všech jeho běžících aplikací. Paměťovou mapu z pohledu obou typu jader lze vidět na obrázku 10. Obrázek zachycuje původní umístění RPMsg buffer deskriptorů. Nekorektní aplikace může zasahovat do paměti jádra a způsobovat těžko laditelné či bezpečnostní problémy. Ochranu paměti je nutné vyřešit na úrovní hardware, který případný přístup na nepovolenou adresu zachytí a neprovede.

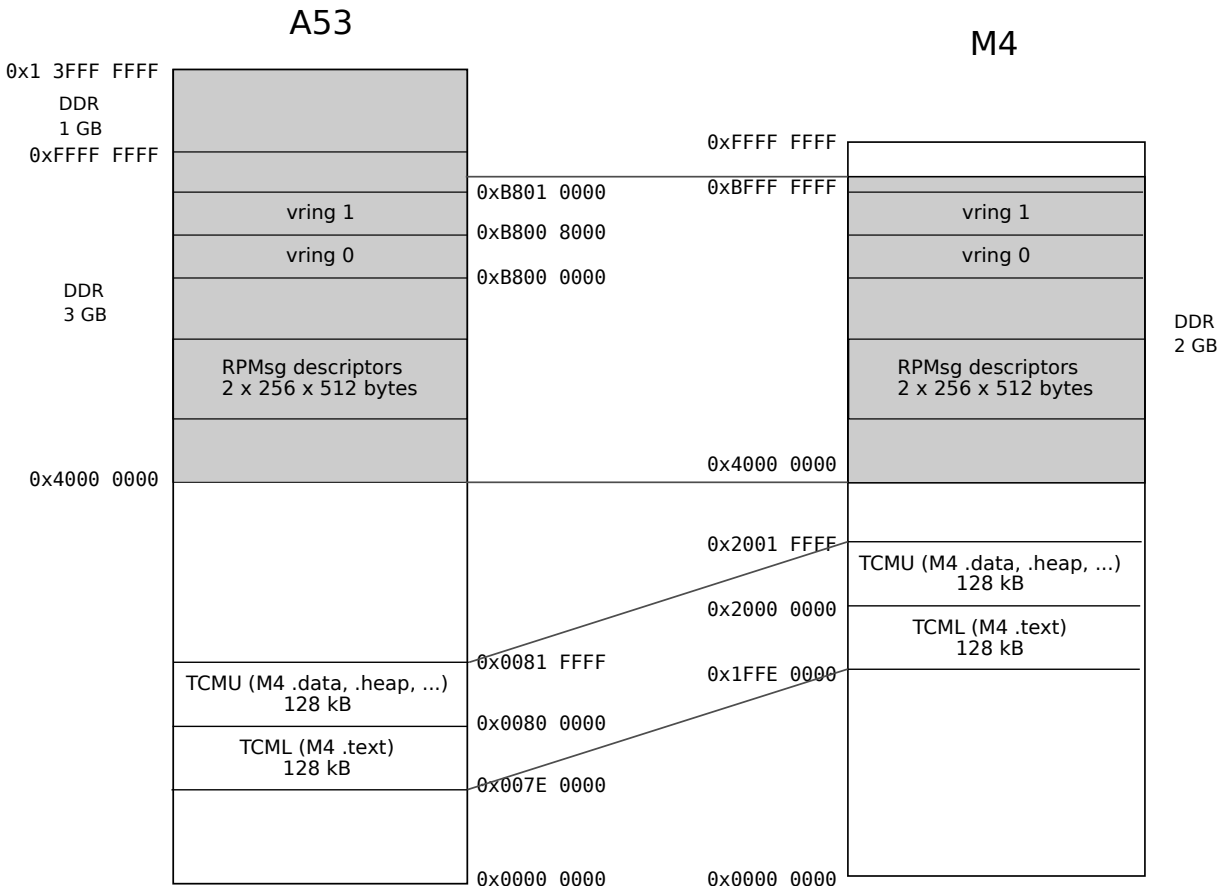
Jedním z možných řešení se nabízí použití Resource Domain Controller, který umožňuje nastavit až 8 regionů s rozlišením 4 kB.

Dalším řešením je použití volitelné ARMv7 komponenty Memory Protection Unit (MPU), která je implementovaná na i.MX 8M procesorech. MPU umožňuje nastavit oprávnění až pro 8 paměťových regionů. Po povolení je každý neoprávněný přístup zachycen a je vyvoláno přerušení, ve kterém lze například aplikaci zastavit.

[32] `https://github.com/TechNexion/u-boot-edm/blob/tn-imx_v2017.03_4.9.88_2.0.0_ga-test/drivers/gpio/mxc_gpio.c#L147`

```
$ echo 119 > /sys/class/gpio/export
$ echo out > /sys/class/gpio/gpio119/direction
$ echo 1 > /sys/class/gpio/gpio119/value
$ m4run rdc_gpio1_m4_only
$ echo 1 > /sys/class/gpio/gpio119/value
[  290.203482] SError Interrupt on CPU3, code 0xbf000002 -- SError
[  290.209417] CPU: 3 PID: 2583 Comm: bash Not tainted 4.9.88-2.0.0
[  290.217510] Hardware name: TechNexion PICO-IMX8M and PI baseboard (DT)
[  290.224042] task: ffff800076986900 task.stack: ffff8000746cc000
[  290.229966] PC is at 0xffff8caa9bac
[  290.233458] LR is at 0xffff8ca55398
[  290.236950] pc : [<0000ffff8caa9bac>] lr : [<0000ffff8ca55398>] pstate:
    20000000
[  290.244348] sp : 0000ffffc33da690
[  290.247666] x29: 0000ffffc33da690 x28: 0000aaaaf509e740
[  290.253008] x27: 0000aaaae502e000 x26: 0000aaaae5041000
[  290.258350] x25: 0000ffff8cb3b648 x24: 0000000000000002
[  290.263691] x23: 0000aaaaf509e920 x22: 0000000000000002
[  290.269033] x21: 0000ffff8cb3b560 x20: 0000aaaaf509e920
[  290.274375] x19: 0000000000000001 x18: 0000ffff8cb3aa70
[  290.279716] x17: 0000ffff8ca52028 x16: 0000aaaae5058bf0
[  290.285058] x15: 0000000000000070 x14: 0000000000000000
[  290.290398] x13: 0000000000000000 x12: 0000000000000020
[  290.295739] x11: 0000ffffc33da638 x10: 0000000000000000
[  290.301081] x9 : 0000ffff8ca559c8 x8 : 0000000000000040
[  290.306422] x7 : 0000000000000001 x6 : 0000000000000031
[  290.311764] x5 : 0000000155510004 x4 : 0000000000000000
[  290.317105] x3 : 0000ffff8cb3f1a8 x2 : 0000000000000002
[  290.322446] x1 : 0000aaaaf509e920 x0 : 0000000000000002
[  290.327785]
[  290.329280] Kernel panic - not syncing: Asynchronous SError Interrupt
[  290.335726] CPU: 3 PID: 2583 Comm: bash Not tainted 4.9.88-2.0.0
[  290.343818] Hardware name: TechNexion PICO-IMX8M and PI baseboard (DT)
[  290.350347] Call trace:
[  290.352805] [<ffff0000080898b0>] dump_backtrace+0x0/0x1b0
[  290.358212] [<ffff000008089a74>] show_stack+0x14/0x20
[  290.363271] [<ffff0000084072c8>] dump_stack+0x94/0xb4
[  290.368330] [<ffff00000817b92c>] panic+0x11c/0x278
[  290.373127] [<ffff00000808a1a0>] do_serror+0x88/0x90
[  290.378097] [<ffff0000080835f8>] el0_error_naked+0x10/0x18
[  290.383586] SMP: stopping secondary CPUs
[  290.388042] Kernel Offset: disabled
[  290.391533] Memory Limit: none
[  290.394596] ---[ end Kernel panic - not syncing: Asynchronous SError
    Interrupt
```

Výpis 17: Fatální chyba při přístupu k nepovolené periférii na straně Linuxu



Obrázek 10: Paměťový prostor z pohledu jádra A53 a M4

Paměť RPMsg byla původně uložena v DDR paměti na dvou místech. Na pevné adrese `0xb800_0000` byla v Device Tree[33] rezervována 1 MB paměť určená pro obě `vring` struktury. Buffery RPMsg se alokovaly z globálního memory pool, který byl při zavádění pomocí parametru `cma` nastaven na první gigabajt DDR paměti. Při takovém stavu by bylo nutné nastavovat ochranu paměti dynamicky. Problémem však je, že buffery pro odesílání z linuxové strany jsou alokovány až při prvním použití. Popsaná úprava v podkapitole 4.3.4 zajistí alokaci `vring` společně s buffery v předem známé ohraničené paměti.

MPU nastavení pro M4 jádro je popsáno ve výpise 18. Nejprve proběhne paměťová bariéra, jejíž úkolem je zajistit, aby se provedly všechny paměťové operace před dokončením této instrukce. Následně dojde k vypnutí MPU. Celkově probíhá nastavení čtyř regionů. V prvních dvou regionech se nastaví plný přístup do TCM paměti. V případě TCMU, obsahující instrukce, se navíc nastaví příznak pro vykonávání. V třetím regionu se povolí plný přístup do paměti registrů periférií. V posledním regionu se povolí přístup do části DDR paměti obsahující obě `vring` struktury společně s buffery. Tato oblast je definovaná v Device Tree konfigurací jako rezervo-

[33] `https://github.com/trnila/linux-tn/blob/tn-imx_4.9.88_2.0.0_ga-test/arch/arm64/boot/dts/freescale/fsl-imx8mq.dtsi#L84`

vaná paměť, kterou Linux využívá pro alokaci při manuálním přiřazení k ovladači `imx_rproc`. Následně proběhne povolení přerušení `MemManage_Fault`, ve kterém lze detekovat nepovolený přístup do paměti.

V předposledním kroku se nastavuje MPU i pro privilegovaný režim. Volitelně lze také nastavit příznak `HFNMIENA`, aby se i přerušení `NMI` (non maskable interupt) a `FAULTMASK` vykonávala s povolenou MPU [17].

V posledním kroku se provádí datová a instrukční bariéra, aby se následující instrukce prováděly s novým nastavením.

M4 jádro umožňuje vykonávání ve dvou režimech - privilegovaném a uživatelském. V privilegovaném režimu jádro vykonává instrukce po restartu nebo v případě přerušení. Jádro může přepnout režim do uživatelského a omezit přístup i k některým instrukcím či nastavení MPU. Pro přepnutí z uživatelského režimu do privilegovaného je zapotřebí provést instrukci `svc` (supervisor call) s číselnou hodnotou, která vyvolá přerušení `SVC` v privilegovaném režimu. Funguje to zde obdobně, jako když userspace aplikace požaduje službu od operačního systému. V případě Linux x86_64 userspace aplikace nastaví číslo systémového volání do registru `rax` a zavolá instrukcí `syscall`, čímž se provede přerušení a požadavek se vykoná v privilegovaném režimu jádra.

FreeRTOS pomocí MPU a uživatelského režimu umožňuje izolovat paměť jednotlivých úloh. Při přepnutí úlohy se kromě obnovy registrů provádí navíc i nastavení MPU. Komunikace mezi úlohami pomocí front či semaforů vyžaduje přepnutí do privilegovaného režimu, což vyžaduje FreeRTOS podporu, která není v rámci SDK podporována. Izolování jednotlivých úkolů je pro využití v rámci této práce zbytečně komplikované a je zcela dostačující ochrana paměti linuxového jádra. M4 jádro vykonává instrukce po celou dobu v privilegovaném režimu. Z tohoto důvodu se nastavuje MPU ochrana primárně pro tento režim.

Je třeba brát na vědomí, že nastavování MPU provádí samotná aplikace. Případná zlomyslná aplikace nemusí nastavit ochranu paměti a může provést například eskalaci oprávnění. Pro demonstraci je vytvořená jednoduchá ukázka `rootkit`, která zasahuje do paměti jádra a konkrétnímu linuxovému procesu změní identifikátor uživatele na root, čímž proces získá vyšší oprávnění. V ukázce se spustí pod normálním uživatelem proces, který čeká, dokud se nepodaří otevřít soubor `/etc/shadow` pro čtení. Při úspěchu nahradí aktuální program za `bash` se získaným root oprávněním. M4 aplikace mezitím v paměti linuxového jádra naivně hledá strukturu `task_struct` dle názvu procesu v proměnné `comm`, kterou předchází dva ukazatele na struktury `cred` obsahující oprávnění daného procesu. Po nalezení stačí transformovat ukazatel z virtuální paměti do fyzické a provést změnu položky `uid` na hodnotu uživatele root. Hodnotu je potřeba zapsat několikrát, protože Linux neočekává, že je daná paměť sdílena. Jakmile se hodnota propaguje na aplikační jádra, tak proces získává root oprávnění a podaří se otevřít soubor `/etc/shadow`. Následně se spustí již zmíněný program `bash` s root oprávněním.

Nutno zmínit, že tato technika může být realizovatelná i na nejnovějším podporovaném jádře verze 4.14.78. V této verzi existuje volba `GCC_PLUGIN_RANDSTRUCT`, která zajistí náhodné

```
void BOARD_InitMemory() {
 __DMB();
 MPU->CTRL = 0;
 /* configure full access to TCMU (128 kbytes)- 2^17 */
 MPU->RBAR = (0x20000000U & MPU_RBAR_ADDR_Msk) | MPU_RBAR_VALID_Msk
           | (0 << MPU_RBAR_REGION_Pos);
 MPU->RASR = MPU_RASR_ENABLE_Msk
           | (0x3 << MPU_RASR_AP_Pos)
           | (1 << MPU_RASR_TEX_Pos)
           | (1 << MPU_RASR_XN_Pos) // not executable
           |(ARM_MPU_REGION_SIZE_128KB << MPU_RASR_SIZE_Pos);

 /* configure full aceess to code TCML (128 kbytes - 2^17) */
 MPU->RBAR = (0x1FFF0000U & MPU_RBAR_ADDR_Msk) | MPU_RBAR_VALID_Msk
           | (1 << MPU_RBAR_REGION_Pos);
 MPU->RASR = MPU_RASR_ENABLE_Msk
           | (0x3 << MPU_RASR_AP_Pos)
           | (1 << MPU_RASR_TEX_Pos)
           | (ARM_MPU_REGION_SIZE_128KB << MPU_RASR_SIZE_Pos);

 /* configure full access to peripherals 16 MB (last 4 MB is reserved) - 2^24*/
 MPU->RBAR = (0x30000000U & MPU_RBAR_ADDR_Msk) | MPU_RBAR_VALID_Msk
           | (2 << MPU_RBAR_REGION_Pos);
 MPU->RASR = MPU_RASR_ENABLE_Msk
           | (0x3 << MPU_RASR_AP_Pos)
           | (1 << MPU_RASR_XN_Pos) // not executable
           | (ARM_MPU_REGION_SIZE_16MB << MPU_RASR_SIZE_Pos);

 /* configure access to RPMsg in DDR (1 MB - 2^20) */
 MPU->RBAR = (0xB8000000 & MPU_RBAR_ADDR_Msk) | MPU_RBAR_VALID_Msk
           | (3 << MPU_RBAR_REGION_Pos);
 MPU->RASR = MPU_RASR_ENABLE_Msk
           | (0x3 << MPU_RASR_AP_Pos)
           | (1 << MPU_RASR_XN_Pos) // not executable
           | (1 << MPU_RASR_S_Pos) // shareable with linux cores
           | (ARM_MPU_REGION_SIZE_1MB << MPU_RASR_SIZE_Pos);

 /* enable MemManage handlers */
 SCB->SHCSR |= SCB_SHCSR_MEMFAULTENA_Msk;
 /* enable mpu and enable MPU for privileged mode also */
 MPU->CTRL = MPU_CTRL_ENABLE_Msk;
 __DSB();
 __ISB();
}
```

Výpis 18: Nastavení ochrany paměti včetně regionu s VirtIO a RPMsg buffery

prohození položek struktur v paměti. Ve výchozí konfiguraci není povolena ani pro současné x86_64 distribuce. Nevýhodou však je, že seed pro prohození položek musí distribuce veřejně poskytnout, jinak by nebylo možné kompilovat out-of-tree moduly [18].

Ve složitější aplikaci může být bez ladících nástrojů obtížné najít místo, kde ke špatnému přístupu do paměti došlo. Z výpisu 19 lze vidět, že se aplikace snažila přistoupit na adresu paměti, kde nemá přístup. Adresu s nepovoleným přístupem lze za určitých podmínek v `CFSR` nalézt v registru `MMFAR` nebo v `BFAR` [17]. V tomhle případě hodnota `0x80` registru `CFSR` určuje, že registr `MMFAR` obsahuje adresu, která způsobila nepovolený přístup. Registr `pc` obsahuje adresu instrukce, která způsobila tuto chybu. Místo ve zdrojovém kódu lze při zkompilovaném program s ladicími symboly nalézt pomocí programu `addr2line` nebo `gdb`.

Vhodnější by však bylo vytvořit výpis paměti (core dump) a ten následně analyzovat pomocí nástroje GDB. V adresáři `debugger/coredumper` je experimentální aplikace, která vytvoří nový *elf* soubor typu `ET_CORE` a vloží do něj jedinou sekci s obsahem paměti TCM. V implementaci chybí registry a není tak možné zjistit poslední instrukci či hodnoty proměnných na zásobníku. Paměť je získaná pomocí zařízení `/dev/mem`. Funkcionalita je také implementovaná v remoteproc novějšího jádra.

V adresáři `debugger/gdbserver` je experimentální implementace `gdbserver.py`. Při chybě `HardFault` jsou registry uloženy na zásobník a adresa zásobníku je následně uložena do globální proměnné. Z této adresy GDB server získá hodnoty registrů.

5.3 Nastavovaní pinů - IOMUX

Některé procesory poskytují více periférií, než mají vyvedených pinů. IOMUX kontrolér nastavuje v době běhu, na které z pevně definovaných pinů bude daná funkcionalita nasměrována. V registru se jedná o položku `MUX_MODE`. Příznak `SION` (Software Input On Field) umožní přístup k vstupní hodnotě bez ohledu na nastavení `MUX_MODE`. Tento příznak musel být nastavený například u periférie UART a I^2C, jinak nebylo možné přijímat žádná data.

Kromě funkcionality lze také nastavit elektrické vlastnosti pinu, jako je proudové omezení, pull-up, open drain atd. Více informaci o možnostech nastavení lze nalézt v aplikační poznámce AN5078 [19].

Nastavení lze provést dle dokumentace nebo pomocí NXP nástroje PINS-TOOL-IMX[34], který generuje C kód nebo Device Tree konfiguraci.

Nastavení funkcionality `GPIO4_IO26` o daných vlastnostech lze vidět ve výpise 20. V Device Tree se do uzlu pojmenovaného `iomuxc` sloučí uvedené nastavení.

5.4 GPIO piny

General-purpose Input/Output (GPIO) pinům lze za běhu nastavit, zda se jedná o pin vstupní či výstupní a následně přečíst jejich logickou úroveň nebo ji také nastavit. Procesor I.MX 8M

[34] `https://www.nxp.com/pages/pins-tool-for-i.mx-application-processors:PINS-TOOL-IMX`

```
MemManage fault
cfsr =      82 mmfar = 40000000 bfar = 40000000
pc = 1ffe713a  sp =       0    lr = 1ffe7831
r0 = 1ffed071  r1 = 20000bc0  r2 = 20000ab4 r3 = 40000000 r12 = 20001c34
$ addr2line -e build/debug/print 1ffe7136
/root/catkin_ws/src/debugger/memmanage/print.c:28 (discriminator 1)
$ gdb build/debug/print
(gdb) list *0x1ffe713a
0x1ffe713a is in consumer_task print.c:28).
23
24      void consumer_task(void *arg) {
25          for(;;) {
26              uint32_t* v;
27              xQueueReceive(Q, &v, portMAX_DELAY);
28              printf("read: %d\r\n", *v);
29          }
30      }
31
32      int main(void) {
(gdb) disassemble 0x1ffe713a
Dump of assembler code for function consumer_task:
0x1ffe7124 <+0>:   push    {r0, r1, r2, lr}
0x1ffe7126 <+2>:   ldr     r5, [pc, #28]  ; (0x1ffe7144 <consumer_task+32>)
0x1ffe7128 <+4>:   ldr     r4, [pc, #28]  ; (0x1ffe7148 <consumer_task+36>)
0x1ffe712a <+6>:   ldr     r0, [r5, #0]
0x1ffe712c <+8>:   add     r1, sp, #4
0x1ffe712e <+10>:  mov.w   r2, #4294967295 ; 0xffffffff
0x1ffe7132 <+14>:  bl      0x1ffe7728 <xQueueReceive>
0x1ffe7136 <+18>:  ldr     r3, [sp, #4]
0x1ffe7138 <+20>:  mov     r0, r4
0x1ffe713a <+22>:  ldr     r1, [r3, #0]
0x1ffe713c <+24>:  bl      0x1ffe077c <printf>
0x1ffe7140 <+28>:  b.n     0x1ffe712a <consumer_task+6>
```

Výpis 19: Možnost jednoduchého zjištění příčiny nepovoleného přístupu do paměti na M4 jádře

```
IOMUXC_SetPinMux(IOMUXC_SAI2_TXD0_GPIO4_IO26, 0U /* no SION */);
IOMUXC_SetPinConfig(IOMUXC_SAI2_TXD0_GPIO4_IO26,
      IOMUXC_SW_PAD_CTL_PAD_DSE(6U) | /* Drive Strength Field - 45 ohm */
      IOMUXC_SW_PAD_CTL_PAD_SRE(3U)); /* Slew Rate Field - max 200 Mhz */
```

(a) Nastavení IOMUX v C

```
&iomuxc {
   pico-8m {
      pinctrl_example: examplegrp {
         fsl,pins = <
            MX8MQ_IOMUXC_SAI2_RXD0_GPIO4_IO23 0x30 /* CAM_SI */
            MX8MQ_IOMUXC_SAI2_TXD0_GPIO4_IO26 0x30 /* CAM_CLK */
         >;
      };
   };
};
```

(b) Nastavení IOMUX v Device Tree konfiguraci

Výpis 20: Nastavení IOMUX v C a v Device Tree konfiguraci

obsahuje celkem 160 pinů rozdělených do tzv. portů po 32 pinech, značené jako `GPIO1`, `GPIO2` až `GPIO5`. Pro využití GPIO funkcionality je nutné nastavit zvolenému pinu funkci `GPIO` v IOMUX.

5.4.1 Periférie GPIO na M4 jádře

Po nastavení pinu na alternativní funkci `GPIO`, lze číst resp. zapisovat logickou úroveň pinu pomocí funkce `GPIO_PinRead`, resp. `GPIO_PinWrite`. Je však nutno zmínit, že může například dojít k souběhu během nastavování logické úrovně pinu na obou jádrech současně, jak bylo vysvětleno v kapitole 5.1. Problémem je, že obě strany nastavují pin pomocí `GPIOx->DR |= 1 << pin`. I když se jedná o jeden řádek, tak zde dochází k načtení aktuální hodnoty `DR` do pracovního registru, který se pomocí operace OR modifikuje a až následně dojde k uložení nové hodnoty zpět do paměti. Během těchto tří instrukcí může dojít k souběhu a tím ke ztrátě nastavení z druhého jádra. Některé procesory, jako má například Raspberry Pi, poskytují dvojici registrů `SET` a `CLEAR`, které umožňují atomické nastavení konkrétní logické úrovně pinu.

V ukázce `gpio_flip.c` jsou demonstrovány 4 metody, jak změnit hodnotu pinu. Přímou úpravou registru bylo možné pin přepnout za 167 ns a při použití funkce `GPIO_PinWrite` ze SDK za 417 ns. Test probíhal s programy, které byly zkompilovány bez optimalizací.

Vstupním pinům lze navíc nastavit, aby při nějaké události, jako je například sestupná hrana, vyvolaly přerušení. Každý GPIO port procesoru I.MX 8M má 2 přerušení. První přerušení `GPIOx_Combined_0_15_IRQn` je vyvoláno, pokud nastane událost na libovolném pinu 0 až 15

a druhé přerušení `GPIOx_Combined_16_31_IRQn` vyvolá přerušení při události na pinech 16 až 31. V rámci přerušení je pak nutné zjistit ze status registru `GPIOx_ISR`, který pin vyvolal přerušení.

V ukázce `gpio_irq.c` se počítá počet sestupných hran pomocí přerušení nastaveném na konkrétním pinu. Během nějaké doby dochází k zastavení počítaní hran. Je to způsobeno linuxovým ovladačem, který modifikuje registr `IMR` (Interrupt Mask Register) a vypíná přerušení pro daný pin, protože zde nikdo o přerušení nezažádal. Vyskytl se také problém, když M4 jádro neodstranilo příznak přerušení. V tomto případě došlo také k opětovnému vyvolávání přerušení i na aplikačním jádře. Pro funkční ukázku je nutné danou periférii zakázat v Device Tree konfiguraci.

Pro demonstraci časově kritické ukázky je připraveno měření vlhkosti ze senzoru DHT11. Senzor je připojen k minipočítači pomocí sběrnice 1-Wire využívající jediný datový signál. Minipočítač zažádá o data nastavením logické 0. Po milisekundě aktivuje pull-up a počká, dokud senzor nenastaví logickou 0 následovanou logickou 1 a poté logickou 0. Po této sekvenci následuje 40bitový přenos, při kterém je logická jednička reprezentována delší dobou nastavené logické 1 oproti 0. Průběh komunikace lze vidět na obrázku 11. Ukázka je implementovaná ve variantě s aktivním čekáním a ve variantě s přerušením vyvolaným změnou hran.

Obrázek 11: Časový průběh komunikace se senzorem DHT11 na sběrnici 1-Wire

5.4.2 Periférie GPIO v Linuxu

Pro ovládání GPIO pinů v linuxovém userspace je stále vhodné zmínit modul GPIO Sysfs Interface i když je již zastaralý. Modul exportuje ovládání pinů v adresáři `/sys/class/gpio/` a díky tomu lze s piny pracovat i v minimalistické BusyBox instalaci. Jednoduchou ukázku pro práci s pinem lze vidět ve výpise 21.

```
$ echo 119 > /sys/class/gpio/export
$ echo out > /sys/class/gpio/gpio119/direction
$ echo 1 > /sys/class/gpio/gpio119/value
$ echo in > /sys/class/gpio/gpio119/direction
$ cat /sys/class/gpio/gpio119/value
```

Výpis 21: Ukázka práce s GPIO pinem pomocí již zastaralého rozhraní

Jedná se o velice jednoduché rozhraní, které lze využít v libovolném programovacím jazyce, který umožňuje práci se soubory. Pomocí systémového volání `poll` nad `value` souborem je možné detekovat např. náběžné hrany[35]. Je však možné, že některé hrany se nepodaří detekovat.

[35] `https://github.com/trnila/picopi8m-ros-demos/blob/master/linux_periph/gpio/sysfs_falling_edge.c`

Velkou nevýhodou tohoto rozhraní je sekvenční značení pinů, dle toho jak byly při zavádění systému nalezeny. V případě, že se v Device Tree konfiguraci zakáže GPIO port, tak dojde k přečíslování a je nutné všechny aplikace překonfigurovat nebo složitě počítat aktuální číslo pinu z informací umístěných v `/sys`.

Novější možnosti ovládání GPIO pinů z userspace je pomocí znakových zařízení `/dev/gpiochipN`, které reprezentují jednotlivé GPIO porty. Znaková zařízení jsou stále číslovaná sekvenčně, i v nejnovější verzi jádra[36], ale implementace konkrétního hardware může portům nastavit popisek. Popisek se v aktuální verzi jádra nenastavuje a proto byla patřičná změna provedena.[37] Dále je v Device Tree konfiguraci dle výpisu 22 možno každému pinu nastavit popisek.

```
&gpio3 {
  gpio-line-names =
    "GPIO3_IO0",
    "GPIO3_IO1",
    "GPIO3_IO2",
    "GPIO3_IO3",
    "GPIO3_IO4",
    "GPIO3_IO5",
    ...
    ;
};
```

Výpis 22: Přiřazení popisků k jednotlivým pinům v Device Tree konfiguraci

Userspace aplikace pak před použitím pinu projde všechny gpiochip zařízení a najde pin dle názvu pomocí volání `ioctl`. Další odlišnosti od GPIO Sysfs Interface je možnost vlastnictví pinu pouze jedním modulem či aplikaci. Aplikace si pomocí `ioctl` volání zažádá o piny ve výstupním směru a výsledkem je nový anonymní souborový deskriptor v předané struktuře, který slouží pro nastavování logické úrovně či jinou operaci. Žádná jiná aplikace či ovladač si nemůže zažádat o pin, dokud je souborový deskriptor otevřený.

Alternativně lze také přiřadit každému GPIO portu deterministický název pomocí *udev* pravidla z výpisu 23. Při nalezení zařízení podle názvu v parametru `KERNELS`, se vytvoří symbolický odkaz z `/dev/gpioN` na `/dev/gpiochipM`. Nastává zde však problém, že některé knihovny či programy umožňují pracovat pouze se soubory `/dev/gpiochipM`.

Další výhodou nového rozhraní je detekce hran bez větších ztrát. Staré rozhraní ukládá pouze informaci[38], že byla detekována hrana. Aplikace není pak schopná zjistit, zda došlo k více

[36] https://github.com/torvalds/linux/blob/a05a14049999598a3bb6fab12db6b768a0215522/drivers/gpio/gpiolib.c#L1279

[37] https://github.com/trnila/linux-tn/commit/19b5ac0977a2f4f2c22d4108327537a3b0a23795

[38] https://github.com/trnila/linux-tn/blob/tn_4.9.88_2.0.0_ga-test-remoteproc/drivers/gpio/gpiolib-sysfs.c#L146

```
# /etc/udev/rules.d/gpio.rules
KERNELS=="30200000.gpio", SYMLINK+="gpio1"
KERNELS=="30210000.gpio", SYMLINK+="gpio2"
KERNELS=="30220000.gpio", SYMLINK+="gpio3"
KERNELS=="30230000.gpio", SYMLINK+="gpio4"
KERNELS=="30240000.gpio", SYMLINK+="gpio5"
```

Výpis 23: Pravidlo *udev* pro vytvoření deterministické cesty k GPIO portům

změnám hran. Nové rozhraní tento problém řeší pomocí fronty[39], do které ukládá při změně hrany čas v nanosekundách. Ve výpise 24 lze vidět, jak získat data z fronty.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/poll.h>
#include <sys/ioctl.h>
#include <linux/gpio.h>
int main() {
   int fd = open("/dev/gpiochip3", O_RDWR);
   struct gpioevent_request gpio_falling;
   gpio_falling.lineoffset = 26; // GPIO4_26
   gpio_falling.handleflags = GPIOHANDLE_REQUEST_INPUT;
   gpio_falling.eventflags = GPIOEVENT_REQUEST_FALLING_EDGE;
   strcpy(gpio_falling.consumer_label, "demo-app-falling");
   ioctl(fd, GPIO_GET_LINEEVENT_IOCTL, &gpio_falling);

   struct pollfd watch = {.fd = fd, .revents = POLLIN};
   while(poll(&watch, 1, -1) > 0) {
      struct gpioevent_data event;
      read(gpio_falling.fd, &event, sizeof(event));
      printf("event %u detected at %llu ns\n", event.id, event.timestamp);
   }
}
```

Výpis 24: Ukázka kódu pro zachytávání náběžných hran v userspace

Používat GPIO v userspace je vhodné až v případě, když neexistuje jaderný modul [20]. Například lze pomocí jádra použít rozhraní 1-Wire vyžadující přesné časování. V subsystému iio (Industrial I/O) je k dispozici ovladač pro teplotní senzor DHT11, který není ve výchozí konfiguraci

[39] https://github.com/trnila/linux-tn/blob/tn_4.9.88_2.0.0_ga-test-remoteproc/drivers/gpio/gpiolib.c#L732

kompilován. Pro zkompilování dynamického modulu je zapotřebí povolit volbu `CONFIG_DHT11=m`. Protože senzor nijak nehlásí svoji přítomnost, je nutné jej nakonfigurovat v Device Tree konfiguraci dle výpisu 25. Podle hodnoty `compatible` ovladač nalezne tento uzel a rezervuje si daný GPIO pin, aby jej nemohl využít jiný ovladač.

Přečíst teplotu či vlhkost lze pomocí přečtení souboru `in_temp_input` resp. `in_humidityrelative_input` v adresáři `/sys/bus/iio/devices/iio:device0/`. Ne všechny požadavky však končí úspěšně, protože se zde vyžaduje přesné časování, které nebylo dodrženo.

```
senzor@0 {
        compatible = "dht11";
        gpios = <&gpio4 26 0>;
        status = "okay";
};

line  24:      unnamed      unused  input active-high
line  25:      unnamed      unused  input active-high
line  26:      unnamed  "senzor@0"  input active-high [used]
line  27:      unnamed      unused  input active-high
```

Výpis 25: Konfigurace senzoru DHT11 v Device Tree konfiguraci a ověření použitého pinu

5.5 Periférie UART

Periférie UART slouží pro nízkorychlostní plně duplexní přenos dat mezi dvěma zařízeními. Procesor i.MX 8M obsahuje 4 UART periférie, z toho je pro aplikaci použitelný za určitých podmínek jenom UART4.

UART1 je použít pro sériovou konzolu Linuxu, která je dostupná jako nultá sériová linka na konektoru microUSB. Konzolu lze uvolnit odebráním `console=ttymxc0,115200` z parametrů jádra. Je ale potřeba brát na vědomí, že zavaděč Das U-boot tuto periférii také využívá a pro úplné uvolnění by byla nutná jeho rekompilace. V případě uvolnění sériové konzoly může být problematické řešení případné nefunkčnosti sítě či samotného zavádění.

UART2 není na minipočítači vyveden. Dle dokumentace [7] je připojen k Qualcomm QCA čipu a použít pro Bluetooth.

UART3 je nakonfigurován jako standardní vstup a výstup M4 jádra připojený na první sériovou linku konektoru microUSB. Na konektor je přivedena funkcionalita z interních pinů, které nejsou vyvedeny. Po modifikaci nastavení IOMUX je možné nasměrovat funkcionalitu i na vyvedené piny.

UART4 je k dispozici pro použití v aplikacích.

5.5.1 Periférie UART na M4 jádře

Pro použití periférie na M4 jádře je vhodné ji zakázat v Linuxu. Souběžné používání vede k chybám linuxového jádra či nedeterministickému chování periférie.

Pro všechny 4 UART periférie jsou připravené varianty s aktivním čekáním, které nastavují IOMUX a hodinový takt. Ukázky slouží pouze pro základní otestování funkčnosti periférie. Další ukázky jsou zaměřeny na periférii UART4, která je volná.

Například ukázka `uart4_readline.c`, využívající FreeRTOS, demonstruje neblokující příjem po řádcích. Aplikace má připravené buffery, do kterých plní přijatá data v přerušení. Při příjmu nového řádku dojde k uložení ukazatele bufferu do fronty, čímž se probudí úloha čekající na další řádek. Úloha převede jednotlivé znaky řádku na velká písmena a odešle je zpět v blokujícím režimu. Po odeslání je ukazatel zařazen do fronty volných bufferů.

5.5.2 Periférie UART na straně Linuxu

Použití periférie UART na straně Linuxu není potřeba detailně rozebírat. Jsou připraveny dvě ukázky. Ukázka `upper.c` čte v blokujícím režimu po znacích a odesílá zpět po převodu na velké písmeno. Druhá ukázka `sender.c` zasílá data po řádcích a následně přečte jeden řádek.

5.6 Periférie SPI

Serial Peripheral Interface (SPI) je full-duplex rozhraní pro komunikaci mezi obvody v roli master-slave, kde master je pouze jeden a iniciuje přenos k jednomu ze slave zařízení identifikované pomocí chip select signálu. Standardně se rozhraní skládá minimálně ze tři signálů - na MOSI vysílá master, na MISO slave a poslední signál SCK určuje hodinový pulz.

Na minipočítači PICO-PI-IMX8M jsou dostupné tři SPI rozhraní, z toho dvě mají vyvedený pouze signál MOSI a SCK, takže jejích použití je limitované write-only přenosem z master zařízení nebo použití 3-wire SPI. Plně dostupné ECSPI1 rozhraní zpřístupňuje chip select pin SS0, který má poněkud složitější nastavení pro transakční SPI, kdy je nutné, aby byl chip select aktivní po celou dobu přenosu.

5.6.1 Periférie SPI na M4 jádře

Pro periférii jsou připravené ukázky v několika adresářích s prefixem `spi` v názvu. V prvním adresáři `spi` jsou blokující varianty pro základní otestování periférie. Dále je připravená ukázka, jak využít softwarový chip select pin a také hardwarový chip select, který je aktivní po celou dobu 128bitového přenosu.

V adresáři `spi_adc128s102` jsou připraveny ukázky pro čtení hodnot z 8kanálového ADC převodníku ADC128S102. Čtení hodnoty kanálu probíhá pomocí 16bitového přenosu, během kterého master zašle číslo kanálu pro další měření. Převodník mezitím zašle zpět naměřenou 12bitovou hodnotu. Z tohoto důvodu některé ukázky zasílají v jednom SPI přenosu kanály

v pořadí: `1, 2, 3, 4, 5, 6, 7, 0`, aby první naměřená hodnota byla z nultého kanálu i během případného resetu převodníku.

Ukázka `adc128s102_sw.c` demonstruje SPI přenos pomocí přepínání logické hodnoty na pinech. Ukázka slouží pouze pro demonstraci a neměla by být reálně použita, protože je značně neefektivní. Další ukázka nazvaná `adc128s102_simple.c` využívá připravenou blokující funkci `ECSPI_MasterTransferBlocking` z SDK. Periodicky se po SPI přenáší jednotlivě naměřené hodnoty kanálu. V ukázce `adc128s102_single.c` je demonstrován přímý přístup do registrů, s tím, že se všechny hodnoty pošlou v rámci jediného SPI přenosu. Obě varianty kontrolují dokončení přenosu aktivním čekáním. Tento způsob by neměl být využit společně s FreeRTOS, protože se během čekání spotřebovává procesorový čas na kontrolování stavu přenosu místo toho, aby se mohla vykonávat jiná úloha. Ukázka `adc128s102_rtos.c` tento problém řeší pomocí SDK funkce `ECSPI_RTOS_Transfer`. Funkce nejprve zamkne mutex, aby nedošlo k použití periférie z jiné úlohy. Následně se provede neblokované nakopírování dat do odesílací hardwarové fronty TXFIFO o maximální velikosti 64×4 bajtů. Po úspěšném nakopírování se funkce pokusí snížit hodnotu semaforu značící dokončený přenos. Běžně bude hodnota semaforu nula a tudíž dojde k zablokování této úlohy do dokončení přenosu. Při dokončení přenosu se v rámci obsluhy přerušení zvýší hodnota semaforu, která také vrací informaci, zda touto operaci došlo k probuzení úlohy s větší prioritou, než je aktuálně prováděna. V takovém případě je na konci obsluhy přerušení úloha přepnuta. Po přepnutí úlohy s SPI přenosem se dokončí operace pro snížení semaforu, odemkne se mutex a volající funkce je informována o úspěšném přenose a přijatých datech. V ukázce v kapitole 7.2 se využívá vlastní implementace společně s přístupem do registrů pro snížení doby měření.

Dále jsou připravené ukázky pro ADC převodník AD1118 a akcelerometr ADXL345.

5.6.2 Rozhraní SPI na straně Linuxu

Na dodaném minipočítači nebylo SPI rozhraní od výrobce v Device Tree popsáno pro konkrétní piny a slave zařízení. V popisu SoC[40] jsou však popsána přerušení a hodiny společně s adresou SPI registrů. Konfiguraci lze rozšířit o nastavení pinů a definici slave zařízení a jejich chip select signálů.

V ukázce z výpisu 26 je definována `ECSPI1` periférie pro dvě slave zařízení `spidev`, kde jedno využívá hardwarový chip select a druhé zařízení využívá softwarový chip select. Zařízení `spidev` je zpřístupněné pro userspace formou souboru `/dev/spidevX.Y`, kde `X` je identifikace periférie a `Y` je slave zařízení či konkrétněji chip select. Nejprve se v uzlu `pinctrl_ecspi1` nastavuje pro zvolené piny jejích alternativní funkce ECSPI1. Hodnota `0x19` nebo binárně `1_1001b` určuje elektrické vlastnosti pinu, které je možno dohledat po jednotlivé piny v dokumentaci. V tomto případě se pomocí spodních `001b` nastavuje proudové omezení pomocí 255 Ω. Předcházející hodnota `1_1000b` určuje maximální rychlost změny logické úrovně na pinu.

[40] `https://github.com/trnila/linux-tn/blob/trn_loadable_imx_rpmsg/arch/arm64/boot/dts/freescale/fsl-imx8mq.dtsi`

Ovladače pro jednotlivá slave zařízení se definují pomocí poduzlů, kde vlastnost `compatible` slouží pro spárování s ovladačem a `reg` určuje, který chip select z vlastnosti `cs-gpios` se využije. Dalším povinným atributem je `spi-max-frequency`. V použitém jádře nefunguje softwarový chip select, protože ovladač nejprve nastaví pin jako výstupní a až následně si o daný pin požádá, čímž se uvede do původního vstupního stavu.

Pro akcelerometr je připravena ukázka `spi_adxl345`. Oproti implementaci na M4 jádře se data akcelerometru přenáší pomocí dvou 32bitových SPI transakcí, protože ovladač neumožňuje nastavit libovolný počet bitů na transakci. Prvním bajtem transakce je adresa registru s příznakem pro sekvenční čtení následujících tří registrů.

5.7 Sběrnice I²C

Nízkorychlostní sběrnice I²C umožňuje připojit až 127 zařízení pomocí signálu na data a hodinový pulz. Na minipočítači je vyvedeno `I2C2` a `I2C3`. Při použití na M4 jádře je ale vhodné periférii na straně Linuxu zakázat. Je také vhodné zmínit, že piny by měly být nakonfigurovány s otevřeným kolektorem s pull-up rezistorem. Dále je dle dokumentace nutné nastavit `SION` bit, aby periférie mohla přistoupit ke vstupu.

Pro M4 jádro jsou připravené ukázky pro teplotní senzor DS1621 a akcelerometr ADXL345. Ukázka použití teplotního senzoru demonstruje variantu s aktivním čekáním. Další ukázka použití senzoru demonstruje využití I²C podpory ve FreeRTOS.

Objevuje se zde ale problém, kdy při okamžitém zahájení dalšího přenosu po aktuálně dokončeném, dochází k uváznutí SDK ve stavu `kCheckAddressState`. Kvůli tomu následující přenosy vrací ihned chybu `kStatus_I2C_Busy`. Problém je způsoben nastavením stavu na `kCheckAddressState` a následném otestování bitu `IBB` v registru `I2Cx_I2SR`. Tento bit vrací logickou 1 při detekované podmínce START na sběrnici. Při detekované podmínce dojde k vrácení chyby `kStatus_I2C_Busy` a zanechání stavu `kCheckAddressState`. Jednoduchým řešením je pokusit se několikrát o zahájení přenosu a při chybě nastavovat manuálně stav na `kIdleState`. Tento problém se může také objevovat při použití multi-master I²C sběrnice.

Na straně Linuxu jsou připravené ukázky s prefixem `i2c` využívající stejné senzory jako na M4 jádře. Neprojevovaly se zde žádné problémy a proto není potřeba tyto ukázky podrobněji rozebírat.

5.8 Časovače a pulzně-šířková modulace

Časovače inkrementují či dekrementují svojí hodnotu dle nastavené frekvence hodinového signálu a při dosažení nastavené hodnoty mohou vyvolat přerušení. Tato funkcionalita je základním prvkem systémů podporujících preemptivní multitasking. Změna kontextu probíhá právě v tomto přerušení.

S časovači souvisí pulzně šířková modulace (PWM), která časovače využívá pro modulaci analogové hodnoty pomocí poměru mezi logickou 1 a 0 na výstupním pinu.

```
&iomuxc {
  pinctrl-names = "default";
  pico-8m {
     pinctrl_ecspi1: ecspi1grp {
        fsl,pins = < MX8MQ_IOMUXC_ECSPI1_MISO_ECSPI1_MISO 0x19
                     MX8MQ_IOMUXC_ECSPI1_MOSI_ECSPI1_MOSI 0x19
                     MX8MQ_IOMUXC_ECSPI1_SCLK_ECSPI1_SCLK 0x19 >;
     };
     pinctrl_ecspi1_cs: ecspi1grp_cs {
        fsl,pins = < MX8MQ_IOMUXC_ECSPI1_SS0_ECSPI1_SS0 0x19 /* hw CS0 */
                     MX8MQ_IOMUXC_NAND_READY_B_GPIO3_IO16 0x19 /* CS 1 */>;
     };
  };
};
&ecspi1 {
  pinctrl-names = "default";
  pinctrl-0 = <&pinctrl_ecspi1 &pinctrl_ecspi1_cs>;
  status = "okay";
  num-cs = <2>;
  cs-gpios = <0>, /* hw CS0 */
             <&gpio3 16 GPIO_ACTIVE_LOW>; /* CS1 */
  spidev@0 {
     compatible = "spidev";
     reg = <0>; /* CS0 */
     spi-max-frequency = <500000>;
     #address-cells = <1>;
     #size-cells = <0>;
  };
  spidev@1 {
     compatible = "spidev";
     reg = <1>; /* CS1 */
     spi-max-frequency = <500000>;
     #address-cells = <1>;
     #size-cells = <0>;
  };
};
```

Výpis 26: Konfigurace dvou SPI slave zařízení `spidev` v Device Tree konfiguraci

Minipočítač obsahuje celkem šest 32bitových časovačů, které jsou volné pro použití na M4 jádře. Časovač vygeneruje přerušení při dosažení hodnoty v některém ze tří output compare registrů. Tato událost může zároveň nastavit hodnotu pinu, které ale nejsou na minipočítači vyvedeny. V kapitole 7.2 je popsán způsob, jak využít output compare registry pro vytvoření dvou synchronizovaných PWM signálů.

M4 jádro má k dispozici časovač nazvaný SysTick, který lze nalézt na všech jádrech ARM Cortex-M. Časovač je ale použít pro preemptivní přepínání úloh ve FreeRTOS.

Procesor i.MX 8M má celkem 4 jednokanálové PWM, ale z toho jsou pouze tři vyvedeny na pinech minipočítače. Ukázka `pwm` demonstruje jednoduché použití všech tří PWM. V ukázce využívá PWM2 zdrojové hodiny o frekvenci 32 kHz běžící i v režimu nízké spotřeby. Další dvě PWM využívají zdrojové hodiny o frekvenci 25 MHz.

5.8.1 PWM na straně Linuxu

Pro použití PWM v Linuxu je nutné nastavit potřebný pin v IOMUX a daný uzel povolit. V ukázce 27 se povoluje periférie PWM3 pro pin `GPIO5_IO3`.

```
#include "pico-8m.dts"
&iomuxc {
   pinctrl-names = "default";
   pico-8m {
      pinctrl_pwm3: pwm3 {
         fsl,pins = <
            MX8MQ_IOMUXC_SPDIF_TX_PWM3_OUT   0x16
         >;
      };
   };
};
&pwm3 {
   pinctrl-names = "default";
   pinctrl-0 = <&pinctrl_pwm3>;
   status = "okay";
};
```

Výpis 27: Device Tree konfigurace pro `PWM3`

Linux povolené PWM zpřístupňuje pomocí sysfs v adresáři `/sys/class/pwm/pwmchipN`. Nastává zde stejný problém při číslování PWM jako u dalších periférií, který lze vyřešit úpravou ovladače `pwm-imx`[41].

[41] `https://github.com/trnila/linux-tn/commit/ed0c5bd10256abd3a43a1b68b20cbc0ff1690728`

Před použitím je nutno pro userspace alokovat jediný PWM kanál `pwm0` zapsáním 0 do souboru `export`. Zapsání vytvoří podadresář `pwm0` obsahující soubory `enable`, `period` a `duty` pro nastavování parametrů.

V ukázkách je demonstrováno jednoduché použití ve skriptovacím jazyce Bash a v jazyce C.

6 ROS a jeho propojení s aplikacemi M4 jádra

Robotický operační systém je userspace framework obsahující knihovny a nástroje, které ulehčují tvorbu robotických systémů. ROS je navrhnut jako modulární systém a jednotlivé uzly (nodes) je možné distribuovat a vykonávat i na jiných počítačích propojených pomocí počítačové sítě. Při správném návrhu aplikace lze přesunout výpočetně náročnější úlohu na výkonnější stroj bez nutnosti upravování aplikace. Komunikace mezi uzly je zajištěna pomocí dvou základních mechanismů.[42]

Prvním mechanismem je komunikace podle vzoru Publish-Subscribe, nazvaná jako témata (topics), která je vhodná pro kontinuální publikování např. naměřených či vypočtených dat dalším uzlům. Publikující uzel (publisher node) publikuje data pevného formátu pod pojmenovanou cestou nazvanou jako téma (topic). Odběratelské uzly (subscriber nodes) se přihlásí k požadovaným tématům pro příjem dat. Publikující a odběratelské uzly nemusí kromě cesty a formátu zpráv o sobě navzájem nic dalšího znát, čímž se snižují závislosti mezi částmi aplikace robotického systému.

Dalším mechanismem je vzdálené volání procedur (remote procedura call - RPC) dle vzoru klient-server, kde klient pošle požadavek a počká na odpověď ze serveru. Tento vzor je v ROS nazýván jako služba (service) a je vhodný pro komunikaci vyžadující výsledek provedené operace.

Dále robotický operační systém obsahuje databázi parametrů, které slouží pro nastavení uzlů. K dispozici je také nástroj pro zachycení publikovaných zpráv a následné přehrání. Dovoluje to reprodukovat případný problém bez nutnosti upravování aplikace. ROS poskytuje seznam[43] s balíčky třetích stran. Balíčkem lze přidat podporu pro detekci obličejů, PID regulátor či Kalmanovo filtrování.

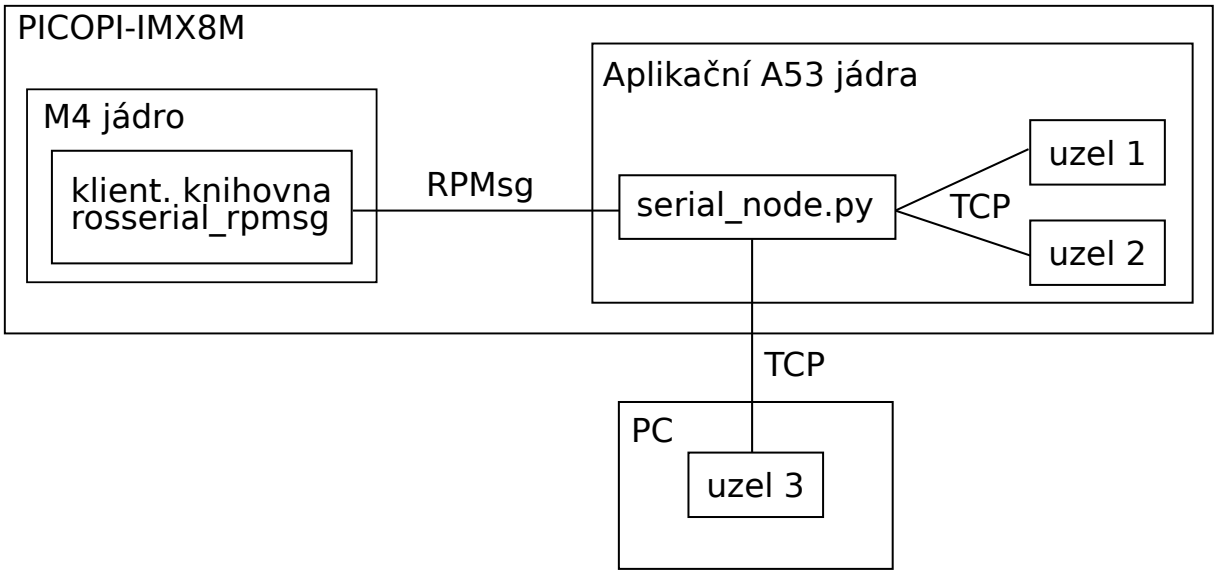
Komunikaci mezi ROS a M4 jádrem lze jednoduše docílit vytvořením uzlu, který se například přihlásí ke konkrétnímu tématu a bude přeposílat data na M4 jádro přes RPMsg ve vlastním formátu. Tento způsob komplikuje vývoj, protože je nutné řešit přenos dat na více místech a je proto vhodné najít řešení, které by umožnilo využít ROS funkcionality přímo na M4 jádře.

6.1 Protokol *rosserial* pro komunikaci mezi ROS a aplikací na M4 jádře

Protokol a knihovna *rosserial* umožňuje využívat ROS funkcionality na procesorech s omezenými prostředky. Protokol definuje, jakým způsobem se mají témata, služby a parametry serializovat do paketů pro přenos skrze komunikační médium, jako je sériová linka či počítačová síť.

Na straně Linuxu se musí vykonávat uzel, starající se o příjem a zasílaní dat na vzdálený procesor. Komunikace mezi uzly a M4 jádrem je znázorněna na obrázku 12. Uzel je naimplementován obecně a není potřeba provádět žádné úpravy pro konkrétní aplikace. V současné době existuje varianta naimplementovaná v jazyce Python a druhá varianta v jazyce C++, která podporuje pouze zasílání a příjem zpráv v rámci témat.

[42] `http://wiki.ros.org/ROS/Tutorials`
[43] `https://index.ros.org/packages/#melodic`



Obrázek 12: Diagram znázorňující komunikaci uzlů s aplikací na M4 jádru

Pro různé platformy a hardware existují tzv. klientské knihovny. Příkladem podporovaných platforem je Arduino, Mbed nebo neoficiální podpora pro procesory rodiny STM32. Klientské knihovny poskytují podobné API rozhraní jako knihovna `roscpp` používaná pro komunikaci s ROS na linuxové straně. Jsou zde mírné odlišnosti jako optimálnější využití staticky alokované paměti pro přenos zpráv. Oproti `roscpp` se pro množinu hodnot téhož typu nevyužívá `std::vector`, ale proměnná s počtem prvků společně s ukazatelem na dané pole.

Podporu pro nový hardware či platformu lze přidat implementací třídy se třemi metodami. První neblokující metoda `read` vrací potencionálně přijatý bajt na komunikačním médiu. Další metoda `write` zasílá paket či obecněji buffer serializovaných dat a poslední metoda `time` vrací čas od startu aplikace v milisekundách.

Klientské knihovny jsou cíleny na aplikace, které nejsou rozděleny do úloh či procesů, ale využívají hlavní smyčku. Nejznámějším zástupcem tohoto paradigmatu je platforma Arduino, kde jedna iterace smyčky reprezentuje kód funkce `loop`. V rámci iterace se sekvenčně provedou jednotlivé úkoly. Obvykle se detekují změny či přijatá data pomocí dotazování (polling). Klientská knihovna očekává, že se metoda `spinOnce` zavolá například na konci jedné iterace. Metoda se pomocí aktivního čekání s nastaveným časovým limitem snaží přečíst data metodou `read` z komunikačního média resp. z bufferu. V případě přečtení celého paketu obsahující zprávu z přihlášeného tématu, dojde k deserializaci a zavolání uživatelské funkce zpracovávající dané téma.

Jak již bylo vysvětleno, aktivní čekání není při použití systému FreeRTOS žádoucí, protože se zbytečně plýtvá procesorovým časem. Dále hrozí riziko, že další události nemusí být zpracovány včas. Některé klientské knihovny podporují částečně FreeRTOS, ale upozorňují, že nejsou concurent-safe [21].

Z tohoto důvodu je metoda `read` upravena na blokující. V případě nedostupnosti dat se zavolá rpmsg-lite funkce, pro příjem RPMsg bufferu, s časovým limitem nastaveným na dvě sekundy. Funkce interně přijímá data z FreeRTOS fronty a tudíž dojde při nedostupnosti dalších bufferů k zablokování úlohy. Pokud do této doby nepřijde RPMsg buffer, vrací se řízení zpět do funkce `spinOnce`, ve které se odešle keep-alive zpráva na uzel běžící na straně Linuxu. Funkce `spinOnce` je cyklicky volaná v separátní FreeRTOS úloze, ve které se vykonává obsluha služeb nebo příjem dat v rámci témat. Obslužné funkce nesmí být blokující či trvat delší čas, jinak mohou zablokovat příjem dalších zpráv či obsluhu služeb. Vhodnější může být přijatá data poslat do fronty a zpracovat v jiné úloze. Po této úpravě dochází k obsluze bez zbytečného prodlení a také se nespotřebovává čas kontrolou, zda je další bajt dostupný.

Odesílané pakety se serializují do globálního bufferu. Pokud budou různé úlohy například publikovat data, tak dojde k souběhu pří zápisu do bufferu. Z tohoto důvodu byla vytvořena třída `FreeRTOSNodeHandle` dědící z třídy `NodeHandle`, která slouží jako centrální přístup k ROS funkcionalitě. Třída přidává globální rekurzivní mutex a překrývá metodu `write`. V metodě se zamkne mutex, zavolá metoda předka a následně se mutex odemkne. Další metody jako přihlášení k odběru či registrace služeb jsou upraveny stejně, aby je bylo možné volat souběžně i v rámci FreeRTOS úloh. Logovací metody, které předávají zprávy do ROS, byly rozšířené o možnost formátování řetězce podobně jako u funkce `printf`. Metoda opět formátuje řetězec do globálního bufferu, protože velikost zásobníku úlohy je podstatně menší, než je zvykem u aplikací běžící na straně Linuxu. Pro zamezení souběhu se před formátováním zamyká mutex a poté se volá metoda `publish`, která také zamyká tentýž mutex. Tento postup by vedl k uváznutí (deadlock) a proto je nutné využít další mutex nebo rekurzivní mutex, který dovoluje stejné úloze provést operaci down, i když je jeho hodnota 0.

V uzlu `serial_node.py`[44], který zajišťuje komunikaci mezi ROS a M4 jádrem, je v rámci práce provedeno několik úprav pro následující problémy. Prvním problémem je, že při zavolání neběžící služby z M4 jádra dochází k pádu uzlu z důvodu neošetřené výjimky. Jednoduchou úpravou je možné tuto chybu zachytit. V současné době u služeb není možné přenášet informaci, zda došlo k chybě při jejím zpracovávání, např. z důvodu neexistující služby. Druhý problém po zavolání služby ze strany Linuxu způsobuje nepředání výsledku. Důvodem je ukládání výsledku do špatné proměnné. Také je zde problém, že čekání na odpověď služby je řešeno pomocí aktivního čekání. Tento nedostatek je vyřešen pomocí `condition variable` s časovým limitem.

Ukázka demonstrující využití ROS bez hardwarových závislostí je popsána v kapitole 7.1. V dalších kapitolách jsou implementované ukázky přistupující k hardware.

6.2 Podpora ROS ve vytvořené distribuci

Pro snadné použití systému ROS a M4 jádra jsou v `/etc/profile.d/` provedená nastavení cest a dalších proměnných, jako je například cesta k MCUXpresso SDK. ROS také hledá balíčky

[44] `$ rosrun rosserial_rpmsg serial_node.py /dev/m4char`

v pracovním catkin adresáři `~/catkin_ws/src` obsahující naimplementované ukázky. Pro kompilaci všech ROS ukázek z pracovního adresáře je nutné spustit příkaz `catkin_make` v adresáři `~/catkin_ws`.

Pro CMake je vytvořena knihovna `rosserial_rpmsg`, která generuje hlavičkové soubory pro ROS zprávy a služby a také klientskou knihovnu *rosserial*. Soubory generované příkazem `rosrun rosserial_rpmsg make_libraries.py` jsou uložené v adresáři `build/ros_lib`. Po modifikaci popisu zprávy či služby je nutné vygenerovat hlavičkové soubory znovu pomocí `make generate_roslib` v adresáři `build`.

Pro finální použití je možné využít `roslaunch`, který dle XML konfigurace spustí potřebné uzly a případně zajistí jejich restartování v případě pádu. Nahrát kód na M4 jádro lze pomocí uzlu `m4ctl` z balíčku `ros_m4ctl`. Konfiguraci pro ukázku z kapitoly 7.1 lze vidět ve výpise 28. Pro vytvořenou konfiguraci lze vytvořit symbolický odkaz vedoucí ze souboru `/etc/roslaunch.xml`. Při startu systému se pomocí vytvořené systemd služby `roscore.service` spustí všechny uzly i s nahráním kódu na M4 jádro. V současné době není automatické nahrání kódu z jaderného modulu remoteproc funkční, protože v době zavedení není připojen souborový systém.

```
<launch>
 <node name="rosserial_rpmsg" pkg="rosserial_rpmsg" type="serial_node.py"
       args="/dev/m4char" />
 <node name="m4_loader" pkg="ros_m4ctl" type="m4ctl"
       args="start $(find ros_m4_demo)/m4/build/debug/ros_m4_demo" />
 <node name="kalman" pkg="ros_m4_demo" type="kalman_filter.py" />
 <node name="publisher" pkg="ros_m4_demo" type="publisher" />
 <node name="addition_server" pkg="ros_m4_demo" type="service_server" />
 <node name="subscriber" pkg="ros_m4_demo" type="subscriber" />
</launch>
```

Výpis 28: Konfigurace `roslaunch` pro spuštění všech uzlů včetně nahrání M4 kódu

7 Ukázky nových možností a ověření spolehlivosti

V této kapitole jsou implementovány ukázky, které zároveň demonstrují použití systému ROS. Ukázky jsou také použity pro otestování časové odezvy či spolehlivosti. Některé ukázky mohou sloužit pro budoucí využití minipočítače pro samořídítelná autíčka.

7.1 Demonstrace využíti ROS s M4 jádrem

Ukázka `ros_m4_demo` demonstruje základní mechanismy ROS a jeho jednoduchou integraci s M4 jádrem pomocí *rosserial*. Základní mechanismy jsou na straně M4 jádra v rámci jedné aplikace, aby byla otestována funkcionalita při souběžném používání. Na straně Linuxu jsou v jazyce C++ připravené jednoduché ukázky, jak konkrétní mechanismus využít. Adresářovou strukturu ukázky lze vidět ve výpise 29.

```
ros_m4_demo
├── m4
│   └── main.c ..........................................................kód M4 jádra
├── src ..................................................................kód pro Linux
│   ├── publisher.cpp .................zasílaní zpráv na M4 jádro v rámci tématu
│   ├── subscriber.cpp ...................příjem zpráv z M4 jádra v rámci tématu
│   ├── service_call.cpp ..............................voláni služby na M4 jádře
│   └── service_server.cpp .......................poskytování služby pro M4 jádro
├── scripts
│   └── kalman_filter.py ............filtrování přijaté polohy do dalšího tématu
├── msg ..................................................definice formátu zpráv
│   ├── Location.msg
│   └── Print.msg
├── srv ..................................................................definice služeb
│   ├── Addition.srv
│   ├── Pause.srv
│   └── SetSigma.srv
├── package.xml
└── CMakeLists.txt
```

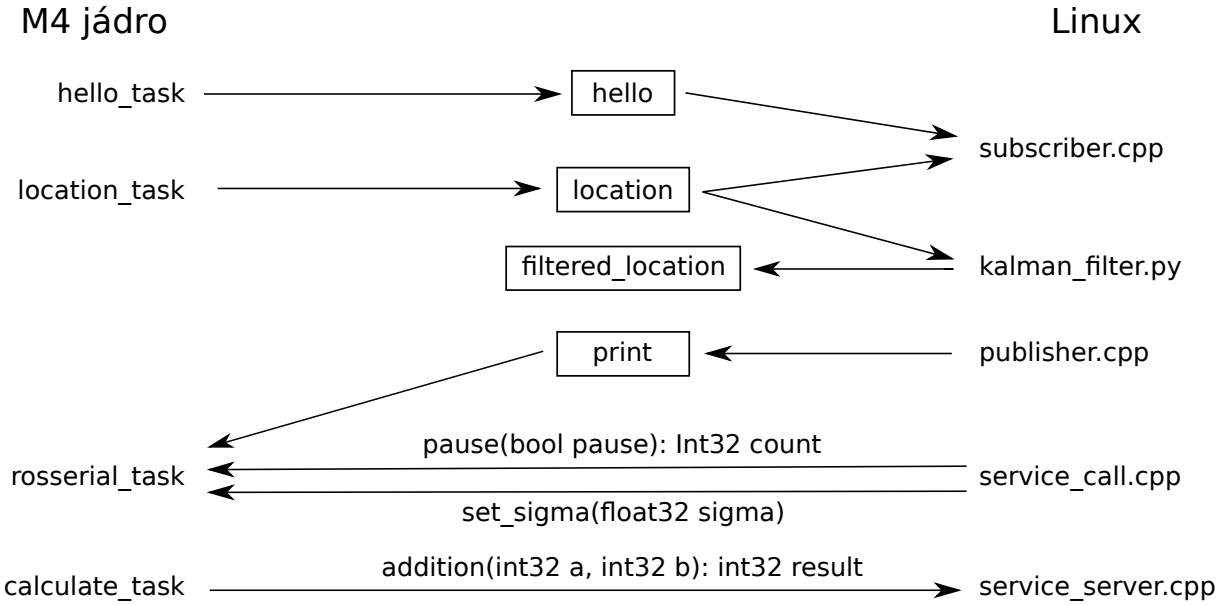
Výpis 29: Adresářová struktura ukázky `ros_m4_demo`

Ukázka na straně M4 jádra je rozdělena do několika FreeRTOS úloh. První úloha `rosserial_task` zajišťuje zpracovávání ROS událostí zaslané z Linuxu pomocí RPMsg. Úloha v cyklu volá metodu `spinOnce` z knihovny *rosserial*. V případě přijaté zprávy v tématu proběhne deserializace a zavolání zaregistrované obslužné funkce. Služby jsou zpracovány obdobně. Zaregistrované funkce se vykonávají v rámci této úlohy, takže blokující obslužná funkce může

zapříčinit pozdější obsluhu dalších ROS události. Další FreeRTOS úlohy publikují data či volají službu na straně Linuxu.

Pro některé ukázky jsou použité vlastní datové typy zpráv či služeb. Typy jsou definované v adresáři `msg` a `srv`. Pro jejich použití je nutné provést kompilaci příkazem `catkin_make` v pracovním adresáři workspace, který je v poskytnuté distribuci v cestě `~/catkin_ws`. Během kompilace se vygenerují hlavičkové soubory či jiná rozhraní pro další jazyky.

V následujícím textu budou popsány jednotlivé mechanismy, jak publikovat zprávu nebo zavolat službu. Na obrázku 13 lze vidět znázorněnou komunikaci. Na levé straně jsou jednotlivé FreeRTOS úlohy a na pravé straně zdrojové soubory jednotlivých uzlů. Obdélníky uprostřed znázorňují témata a zbytek jsou služby. Názvy jsou zde uvedeny bez prefixu `/ros_m4_demo/`. Ukázky použití nástrojů pro komunikaci lze vidět ve výpise 30.



Obrázek 13: Komunikace mezi M4 jádrem a Linuxem pomocí ROS mechanismů

M4 jádro zasílá data do Linuxu publikováním zpráv v tématu `/ros_m4_demo/hello`. Publikování je realizováno ve FreeRTOS úloze `hello_task`. Přijímat zprávy na Linuxu lze pomocí příkazu `rostopic echo /ros_m4_demo/hello` nebo pomocí připravené ukázky `subscriber`.

Příjem dat z Linuxu je realizován pomocí odběru zpráv z tématu `/ros_m4_demo/print` s datovým typem `ros_m4_demo/Print`, který je definován v `msg/Print.msg`. Přijatá zpráva složená z textového řetězce a čísla je na straně M4 jádra vytisknuta na konzolu v rámci vykonávání úlohy `rosserial_task`. Odeslat data lze z Linuxu příkazem `rostopic pub` nebo připravenou ukázkou `publisher`.

Služba typu `ros_m4_demo/Pause` na straně M4 jádra dle parametru uspí nebo probudí FreeRTOS úlohu, která zasílá generovanou polohu. Služba vrací proměnnou s počtem provedených uspání resp. probuzení. Z Linuxu lze službu s parametry zavolat pomocí příkazu `rosservice call ros_m4_demo/pause false` nebo pomocí připravené ukázky `service_call`.

```
$ rostopic echo /ros_m4_demo/location
position: -0.653825759888
velocity: 0.387439608574
acceleration: 0.687521517277
real_position: -0.992185890675
---
position: -1.13367772102
velocity: -0.479851961136
acceleration: -0.86729156971
real_position: -0.974773049355
...
$ rosservice call ros_m4_demo/set_sigma 7.5
$ rostopic pub /ros_m4_demo/print ros_m4_demo/Print "Hello World" 64
```

Výpis 30: Ukázka použití ROS nástrojů pro komunikaci s M4 jádrem

Naopak úloha `calculate_task` periodicky volá službu `ros_m4_demo/Addition` na straně Linuxu. Vstupem jsou dvě čísla, které se na Linuxu sečtou a na M4 jádro se vrátí výsledek.

V úloze `location_task` je simulován senzor smyšlené polohy jako sinusový signál zatížený Gaussovým šumem, který lze nastavit pomocí služby `ros_m4_demo/SetSigma` běžící na straně M4 jádra. Ukázkou konzumenta této smyšlené polohy je uzel provádějící filtrování pomocí Kalmanova filtru napsaného v jazyce Python. Uzel přijímá zašuměná data z tématu, provádí jejích filtraci a následně publikuje vyfiltrovaná data do dalšího tématu.

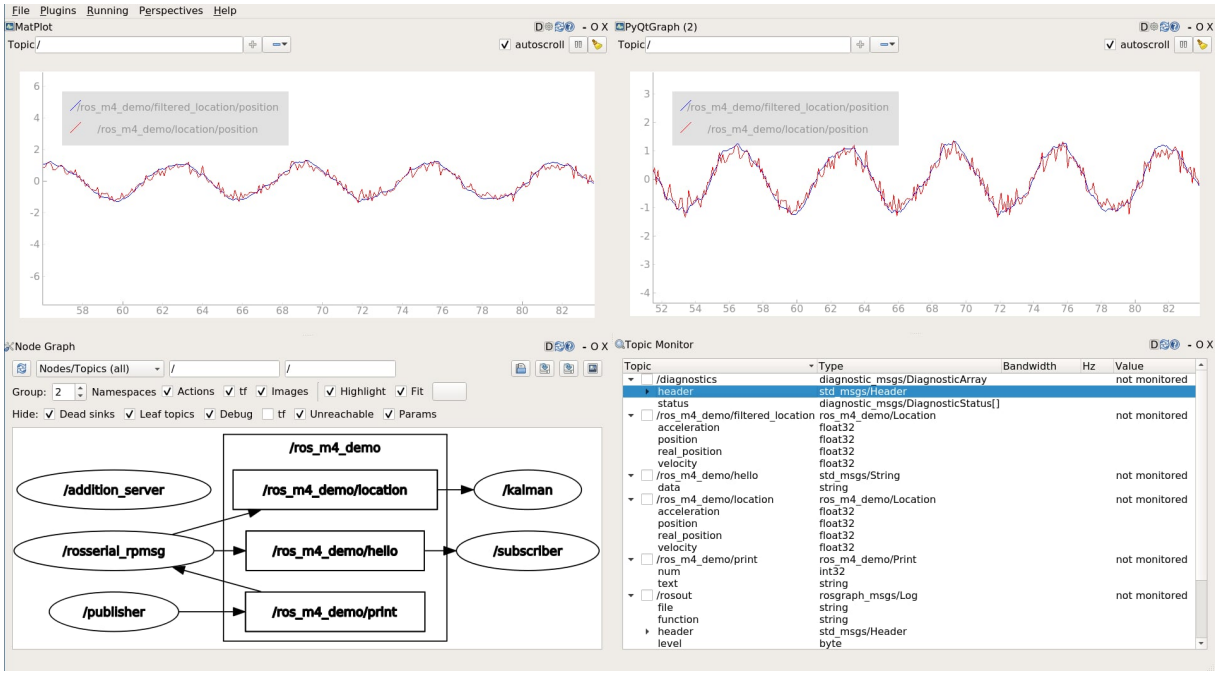
Pro jednoduchou tvorbu grafických rozhraní slouží framework `rqt` naprogramovaný v jazyce Python za použití grafických komponent z Qt. Existuje několik předpřipravených rozšíření jako `rqt_plot` vykreslující graf z publikovaných dat v tématech, či sledování množství publikovaných dat v tématech nebo také graf znázorňující tok dat. Ukázku použití lze vidět na obrázku 14. Pro spuštění na počítači lze využít připravený Docker kontejner z repositáře `picopi8m-rosclient`[45]. V kontejneru lze spouštět i X11 aplikace bez nutnosti instalace ROS do systému. Je však nutné provést kompilaci zpráv a služeb v kontejneru pomocí `catkin_make`.

7.2 Pořizování snímků pomocí analogové řádkové kamery

Ukázka `linecam` postupně pořizuje snímky z analogové řádkové kamery TSL1401R-LF [22], která se využívá pro samořiditelná autíčka. Kamera se skládá ze 128 fotodiod snímajících intenzitu světla. Při hodinovém pulzu CLK a logické jedničce na pinu SI se provede měření všech fotodiod a následné uložení do posuvného registru. Analogové hodnoty pixelů lze z posuvného registru postupně získat pomocí hodinového pulzu CLK.

Procesor i.MX 8M neobsahuje ADC převodník a proto je nutné využít externí ADC převodník připojený pomocí sběrnice I^2C nebo SPI rozhraní. Nevýhodou I^2C převodníků je však

[45] `https://github.com/trnila/picopi8m-rosclient`



Obrázek 14: Ukázka nástroje rqt

nízká maximální frekvence sběrnice 400 kHz a proto byl vybrán 8kanálový ADC převodník ADC128S102 na SPI rozhraní s maximální frekvencí 16 MHz.

Schéma zapojení ukázky lze vidět na obrázku 15. SPI rozhraní včetně hardwarového chip select pinu minipočítače je připojeno na rozhraní ADC převodníku. Nultý kanál převodníku je připojen na analogový výstup řádkové kamery. Dva signály `CAM_SI` a `CAM_CLK` vedoucí k řádkové kameře jsou na minipočítači nastavené jako výstupní GPIO piny.

Pro přečtení nového řádku procesor nastaví logickou jedničku na SI pin a provede hodinový takt na pinu CLK. Následně na SI pinu nastaví logickou nulu. Poté se pro všechny pixely postupně provede další takt a zahájí SPI přenos o velikosti 16 bitů pro přečtení hodnoty pixelu.

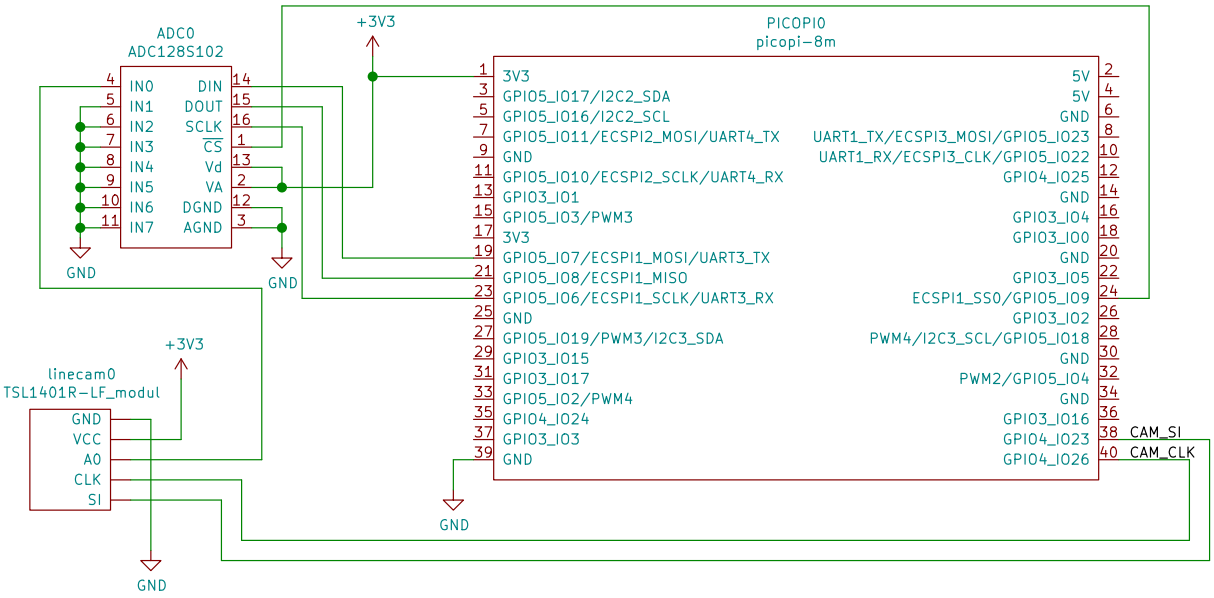
V následujících podkapitolách jsou popsané a implementované různé způsoby od využití M4 jádra až po vlastní modul v operačním systému. U některých ukázek jsou i zobrazeny časové odezvy mezi dokončením a zahájením SPI přenosu.

Všechny ukázky poskytují naměřená data v ROS tématu pro následné zpracování. Jednou z ukázek je OpenCV či webová aplikace zobrazující obraz složený z posledních řádků.

7.2.1 Pořizování snímků na straně M4 jádra

Pořízení snímku je na M4 jádře realizováno třemi způsoby ve vlastní FreeRTOS úloze. Po pořízení snímku se zvýší hodnota semaforu, čímž se probudí úloha odesílající snímek do ROS a pořizovací úloha se uspí.

První jednoduchá varianta `linecam_simple` provádí SPI přenos pomocí funkce s aktivním čekáním. Jedná se o srozumitelnou ukázku, která není efektivní v rámci FreeRTOS.



Obrázek 15: Schéma zapojení ukázky `linecam`

Druhá varianta `linecam_irq` využívá FreeRTOS podporu pro periférii SPI. Pro změření dalšího řádku se nastaví `CAM_SI` signál a zahájí SPI přenos naměřené hodnoty na ADC kanále 0. Po zahájení přenosu je aktuální úloha uspána. Po dokončení přenosu je z přerušení vyvolána funkce. Pokud se jedná o poslední pixel, tak je zvýšena hodnota semaforu pro probuzení odesílací úlohy. V opačném případě se provede hodinový takt `CAM_CLK` a zahájí přenos následujícího pixelu.

Využití SDK v předchozí variantě přináší režii, která se promítá do latence. Z tohoto důvodu ukázka `linecam_raw` přistupuje do registrů a obsluhuje SPI přerušení bez využití SDK. Obrázek 16 znázorňuje časový průběh začátku měření. Doba mezi SPI přenosy při přímém přístupu k registrům (obrázek 16b) je přibližně dvakrát menší než při použití funkcí z SDK (obrázek 16a). Pořízení celého řádkového snímku v ukázce `linecam_raw` trvá přibližně 340 µs.

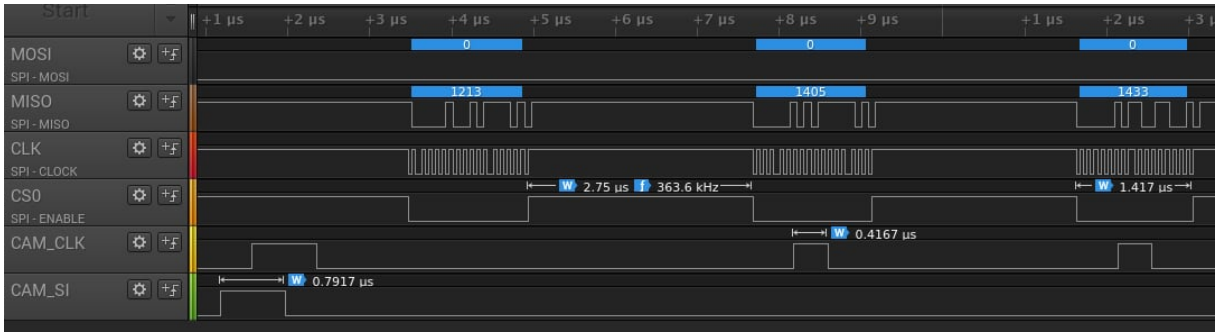
7.2.2 Pořizování snímků na straně linuxového userspace

Pro porovnání časové odezvy je vytvořená ukázka `linecam_linux` běžící v linuxovém userspace. Ukázka vyžaduje nastavení GPIO pinů a rozhraní SPI v Device Tree konfiguraci `pico-8m-linecam.dts`, která je součásti zdrojových kódu linuxového jádra.

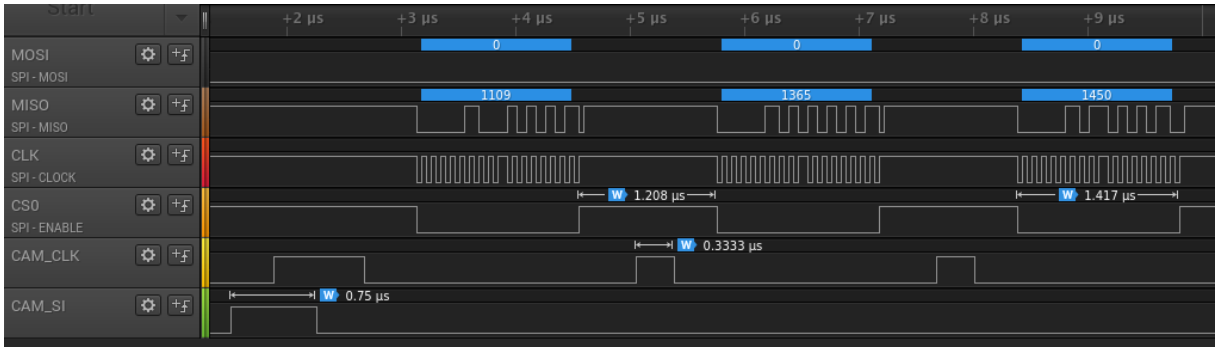
Logická hodnota GPIO pinů se nastavuje pomocí novějšího GPIO rozhraní `gpiolib`. SPI komunikace využívá modul `spidev`. Před začátkem komunikace je nutné nastavit délku SPI přenosu pomocí `ioctl`.

Na obrázku 17a lze vidět, že časová odezva mezi dokončením a zahájením SPI přenosu je přibližně 195 µs. Nutno připomenout, že pořízení celého snímku z M4 jádra trvá přibližně 340 µs.

Pouhé nastavení vyšší priority nepřinese žádné významné zkrácení doby odezvy. Zkrátit dobu lze vykonáváním procesu na prvním jádře, které obsluhuje přerušení periférií. Tuto skutečnost lze ověřit pomocí souboru `/proc/interrupts`, který zobrazuje čítače přerušení na jednotlivých



(a) Ukázka `linecam_irq` využívající SDK



(b) Ukázka `linecam_raw` s přímým přístupem k periferiím

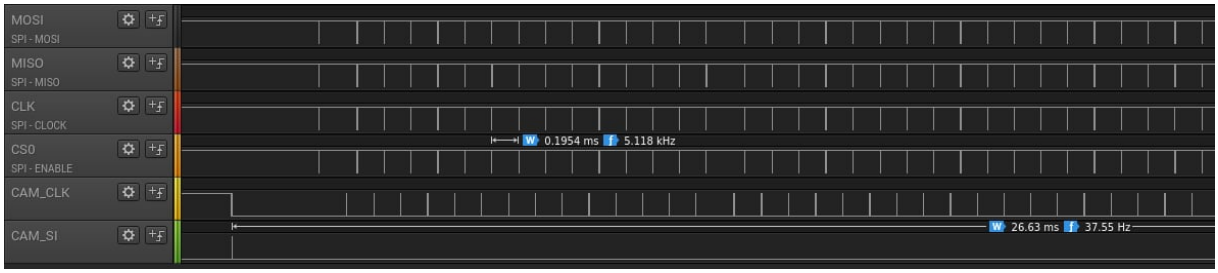
Obrázek 16: Časové odezvy ukázek `linecam` na M4 jádře

jádrech. Plánovat proces na konkrétní jádro lze pomocí programu `taskset`. Jádro lze navíc pomocí jaderného parametru `isolcpus` izolovat z plánovače, aby pro něj nebyly plánované jiné procesy. Na obrázku 17b lze vidět, že po těchto nastaveních se doba odezvy může snížit až na 48 ns. Na obrázku lze také vidět, že odezva není deterministická - doba mezi SPI přenosy někdy trvala i 128 ns. Nevýhodou této úpravy je vyhrazení aplikačního jádra pro časově kritické zpracovávání, které není ani možné při běhu systému Linux deterministicky zajistit.

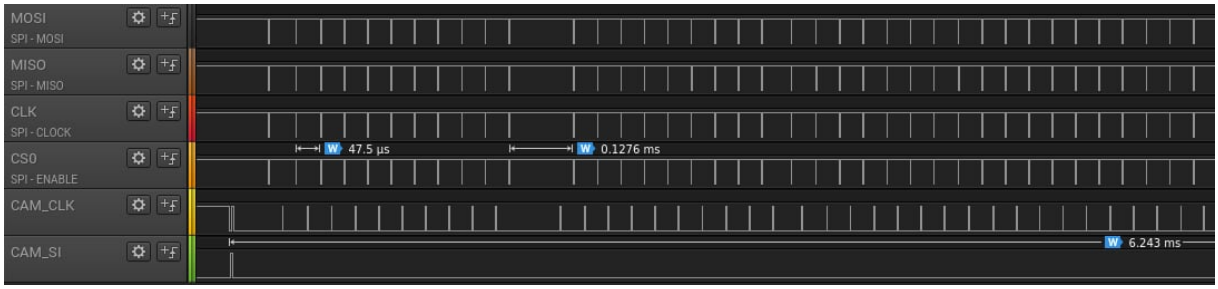
7.2.3 Pořizování snímků v jaderném modulu

Pro zajímavost a porovnání odezvy je vytvořená ukázka modulu `linecam.ko` běžící přímo v jádře operačního systému. Ukázka vyžaduje připravenou Device Tree konfiguraci `pico-8m-linecam-kernel.dts`. V konfiguraci se zakáže uzel s ovladačem `spidev` a nahradí se uzlem pro ovladač `linecam.ko`. Dále jsou v uzlu popsány GPIO piny pro signály `LINECAM_SI` a `LINECAM_CLK`. Po této úpravě se při zavedení modulu zavolá funkce `probe` SPI ovladače `linecam.ko`. Během inicializace se alokují GPIO piny z Device Tree konfigurace a vytvoří se znakové zařízení sloužící pro komunikaci s userspace.

Userspace aplikace pro pořízení snímku otevře znakový soubor `/dev/linecam0` a zavolá systémové volání `read` s předaným bufferem o velikosti jednoho snímku. Ovladač po naměření celého snímku překopíruje data do userspace pro další zpracování.



(a) Běh procesu na libovolném aplikačním jádře



(b) Běh procesu s nejvyšší prioritou na izolovaném aplikačním jádře

Obrázek 17: Časové odezvy ukázek `linecam` na aplikačním jádře v userspace

Na obrázku 18 lze vidět, že pořizování v jádře snížilo odezvu mezi dvěma SPI přenosy na 30 µs. Lze také vidět, že nebylo zajištěno deterministické časování, i když bylo dané jádro izolováno z plánovače a nastaveno na nejvyšší prioritu.

Nevýhodou tohoto způsobu je větší složitost vývoje, která může vést k nestabilitě celého systému. Například při prioritním vykonávání na prvním jádře dochází ke zpožďování obsluhy Ethernetu.

Obrázek 18: Časová odezva ukázky `linecam` běžící v jaderném modulu

7.3 Časové odezvy a propustnost při komunikaci mezi jádry

RPMsg komunikace mezi jádry probíhá výměnou bufferů o maximální délce 496 bajtů. Pro spočítání maximální datové propustnosti modelu komunikace request-response, je potřeba zjistit časovou odezvu mezi zasláním zprávy a získáním odpovědi.

Nejprve je vhodné otestovat odezvu na minimalistickém příkladu, který mezi jádry posílá data pouze pomocí Messaging Unit. M4 jádro v ukázce `mu.c` povolí MU přerušení, ve kterém přijatou zprávu inkrementuje a pošle zpět. Na straně Linuxu je připraven modul `mu_bench.ko`,

který zaregistruje obslužnou funkci pro MU přerušení. V obslužné funkci se při příjmu zprávy uloží uplynulý čas od posledního odeslání zprávy. Obsah zprávy se následně inkrementuje, a pokud je menší než počet pokusů, tak se pošle na M4 jádro pomocí MU. Celé měření odezvy se odstartuje pomocí přečtení znakového zařízení `/proc/mu_bench`, které je blokující, dokud se neprovede celé měření.

Pro ovladače `rpmsg_m4char` a `imx_rpmsg_tty` je připraveno měření, které zasílá mezi M4 jádrem a userspace zprávu o délce 4 B. Doba odezvy mezi zasláním a přijetím odpovědi se měří obdobně.

V porovnání `rostopic` se z linuxového procesu naimplementovaného v jazyce C++ publikuje zpráva do tématu. Tato zpráva je přijata uzlem `serial_node.py`, serializovaná a zaslána po RPMsg na M4 jádro. Jádro zprávu příjme a publikuje inkrementovanou hodnotu do tématu, jehož odběratelem je linuxový proces. Doba mezi odesláním a příjmem zprávy je změřena.

V tabulce 3 lze vidět časové odezvy pro 10 000 požadavků. Doba odezvy mezi prvním a třetím kvartilem je pro MU a ovladače stabilní. U ovladače `imx_rpmsg_tty` si lze povšimnout, že pouhých 100 odpovědí trvalo déle než 256,93 µs. Nejdelší odezva byla až 2637,96 µs. Publikování v tématech přidává značnou režii, která může být způsobena neefektivní implementaci v jazyce Python. Měření bylo provedeno na nezatíženém systému, bez nastavených priorit a s programy zkompilovanými bez optimalizací.

Teoreticky při použití `rpmsg_m4char`, modelu request-response a velikosti dat 496 B lze dosáhnout střední propustnosti až 3,5 $\mathrm{MB\,s^{-1}}$ v rámci jednoho směru. Střední počet zaslaných zpráv mezi jádry je 7 434 za sekundu. Při použití ROS je to pouze 262 zpráv za sekundu.

Tabulka 3: Doba odezvy v µs při zasílání zpráv mezi jádry modelem request-response

kvantil	**mu_bench**	**rpmsg_m4char**	**imx_rpmg_tty**[46]	**rostopic**
0%	4,44	89,88	127,44	2 176,90
25%	7,80	134,28	190,32	3 368,46
50%	7,80	134,52	190,80	3 819,33
75%	7,92	134,64	192,48	4 531,22
99%	9,60	193,80	256,92	5 960,12
100%	71,04	263,16	2 637,96	7 393,13

Propustnost bez modelu request-response je závislá na konkrétní implementaci. Přijaté zprávy jsou z M4 jádra v rámci ovladače `rpmsg_m4char` kopírovány do bufferů o maximální délce 10 000 zpráv. Další zprávy jsou následně zahazovány.

7.4 Použití PWM a časovačů pro řízení motorů

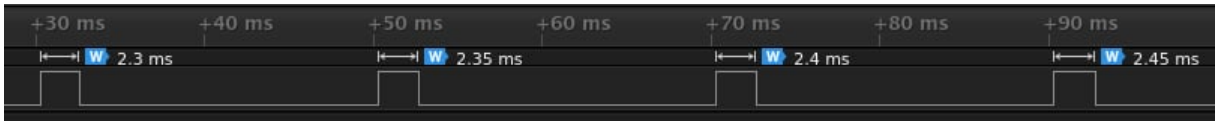
Ukázka `motors` demonstruje ovládání dvou stejnosměrných motorů a dvou servo motorů pro budoucí použití na samořiditelných autíčkách. Na minipočítači jsou vyvedené pouze tři jednoka-

[46] modul zkompilovaný s `DYNAMIC_DEBUG=n`

nálové PWM periférie. Z tohoto důvodu je PWM využito pouze pro serva u kterých se nastavuje pulz v rozmezí 500 µs až 2500 µs.

Stejnosměrné motory se ovládají pomocí časovače `GPT1` o třech kanálech. Po nastavení potřebných děliček se inkrementuje 32bitový časovač s frekvencí 40 MHz. Časovač vyvolá přerušení, jakmile dosáhne hodnoty na některém z kanálu. První kanál je nastaven na frekvenci 4 kHz a reprezentuje periodu pulzně-šířkové modulace. Při jeho přerušení se nastaví logická úroveň 1 na oba piny servo motorů. Další dva kanály určují střídu serv - v rámci přerušení se nastaví logická úroveň 0.

Ukázka je integrována do robotického operačního systému pomocí odběratelských témat na straně M4 jádra. Každý motor lze ovládat pomocí samostatného tématu. Hodnoty všech motorů lze také nastavit „atomicky“ pomocí tématu s datovým typem `carmotor_msgs/CarMotor`. Nastavení motorů je prováděno `bash` skriptem, který generuje ROS zprávy a následně zasílá rourou do programu `rostopic`. Tento program postupně publikuje zprávy do M4 jádra s frekvencí 50 Hz. Časový průběh jednotlivých nastavení lze vidět na obrázku 19.



Obrázek 19: Časový průběh při nastavování servo motoru z ROS

7.5 Počítání hran na GPIO pinech

V ukázce `gpio_interrupts` zaznamenává M4 jádro počet hran na jednotlivých pinech pomocí přerušení. Pro počítaní hran na pinu je nutné nejprve zavolat službu `/gpio_interrupts/enable`, které se předá číslo portu, pin a zda se má počítat náběžná či sestupná hrana nebo také obojí. Počty hran jsou následně publikovány do tématu `/gpio_interrupts/counts`. Ukázka běžící na straně Linuxu registruje dva GPIO piny a následně vypisuje počty hran.

8 Závěr

Cílem práce bylo vybrat vhodný minipočítač s čtyřjádrovým procesorem i.MX 8M a upravit jeho programové vybavení tak, aby bylo možné využívat ROS a současně pro real-time aplikace jeho mikropočítačové jádro Cortex-M4F. Oproti jednojádrovému minipočítači BeagleBone Black, který obsahuje dvě real-time jádra pro obsluhu periférií, se zde primárně využívají hardwarové periférie. Díky tomu je například možné obsluhovat více periférií současně. Pro procesor i.MX 8M byl vybrán cenově dostupný minipočítač PICO-PI-IMX8M malých rozměrů s dostatečným množstvím vyvedených pinů. Na úkor těchto požadavků však nebyla dostupná kompletní dokumentace či softwarová podpora pro snadné použití a vývoj.

Dodaná distribuce Yocto komplikovala vývoj robotických aplikací, protože bylo zapotřebí kompilovat všechny programy včetně závislosti. Z tohoto důvodu byla v rámci práce vytvořena vlastní distribuce založená na Ubuntu. Distribuce poskytuje velké množství předkompilovaných balíčků včetně balíčků pro robotický operační systém.

Podporované linuxové jádro neumožňovalo nahrávat nový kód pro M4 jádro a bylo tedy nutné nahrávat kód buď ze zavaděče operačního systému nebo pomocí userspace aplikace, která přistupuje k registrům a paměti M4 jádra na přímo. Nahrávaní kódu ze zavaděče není vhodné pro vývoj, neboť bylo zapotřebí opakovaně startovat operační systém i s uzly robotického operačního systému. Nahrávání pomocí userspace aplikace způsobovalo, že VirtIO struktury, použité pro komunikaci mezi jádry, byly při opakovaném startu M4 jádra v nekonzistentním stavu. Tento stav vedl k opakovanému přijetí starých zpráv, k postupné ztrátě všech bufferů nebo v horším případě k nefunkčnosti aplikace, která vedla k nutnosti restartovat minipočítač. Výše zmíněné problémy byly vyřešeny pomocí jaderného frameworku remoteproc, který umožňuje nahrávat, ovládat a spravovat zdroje dalších jader systému. V rámci práce byl začleněn jaderný remoteproc modul `imx-rproc` a přizpůsoben pro procesor i.MX 8M. Modul byl dále rozšířen o podporu zasílání zpráv pomocí spravovaného VirtIO zařízení, takže při opakovaném startu M4 jádra dochází k uvedení VirtIO struktur do konzistentního stavu. Novější linuxové jádro může do budoucna přidat podporu pro coredump či jednoduché poskytnutí obsahu paměti M4 jádra do userspace.

Aplikační jádra spolu s mikropočítačovým jádrem v jednom pouzdře přináší několik problémů, jako je souběh při přístupu k periferiím či modifikace paměti jiného jádra. Dynamické uvolnění periférií není možné, protože některé ovladače tuto možnost neberou v úvahu. Proto je stále nutné v Linuxu zakázat periférie v Device Tree konfiguraci před samotným zavedením linuxového systému. Dalším problémem je, že M4 jádro má plný přístup do části DDR paměti, ve které je linuxové jádro spolu s uživatelskými procesy. Chyba v programu může vést k fatálním chybám vedoucí například i k přepsání obsahu souborů. Pro zamezení problémů byla v rámci práce nakonfigurována jednotka správy paměti, kdy M4 jádro může pouze do DDR paměti s VirtIO strukturou a RPMsg buffery. Pro tuto funkcionalitu musel být také upraven linuxový modul, aby alokoval RPMsg buffery v ohraničené paměti. Na ukázce běžící na M4

jádře je demonstrováno, jak bez ochrany paměti eskalovat oprávnění linuxového procesu. Avšak jednotka správy paměti tento problém neřeší, ale pouze zamezuje nechtěné modifikaci linuxové paměti způsobenou chybou v programu M4 jádra. Tento problém by mohl být vyřešen například nastavením přístupu do paměti pomocí Resource Domain Controller, který umožňuje zamknout nastavení do dalšího restartu procesoru.

Pro robotické aplikace existuje framework nazvaný robotický operační systém, který poskytuje mechanismy pro komunikaci mezi uzly. Cílem práce bylo využít M4 jádro pro real-time část aplikace, která komunikuje s ostatními uzly vykonávajícími se na aplikačních jádrech s větším výpočetním výkonem. V rámci práce byly naimplementovany ukázky použití jednotlivých periférií pomocí různých způsobů od neefektivního aktivního čekání až po variantu s obsluhou v přerušení s real-time jádrem FreeRTOS. Pro operační systém Linux byly také naimplementovány ukázky použití periférií avšak v menší míře. Pro M4 jádro byla v rámci práce implementována podpora pro knihovnu *rosserial*, díky které se M4 jádro stává součástí ROS a může tak komunikovat s ostatními uzly robotického systému. Knihovna cílí na platformy Arduino, a proto bylo potřeba provést úpravy pro efektivnější běh ve FreeRTOS. V rámci práce bylo provedené měření časové odezvy při komunikaci mezi jádry. Zaslání 32bitové zprávy mezi M4 jádrem a Linuxem se pohybovalo v jednotkách mikrosekund. Komunikace mezi userspace a M4 jádrem za použití naimplementovaného ovladače `rpmsg_m4char` dosáhlo střední časové odezvy přibližně 134 µs. Oficiální ovladač `imx_rpmsg_tty` dosahoval střední časové odezvy přibližně 191 µs, ale byly naměřeny i časové odezvy dosahující jednotky milisekund. Střední časová odezva mezi uzlem robotického operačního systému a M4 jádrem dosáhla přibližně 3819 µs. Vyšší odezva byla způsobena neefektivní komunikaci prostřednictvím uzlu, který obstarává komunikaci mezi ROS a M4 jádrem.

Pro demonstraci využití M4 jádra byla vytvořena ukázka zachycující snímky na řádkové kameře. Časové odezvy byly porovnány mezi aplikaci vykonávající se na M4 jádře, userspace a v linuxovém modulu. Aplikace na M4 jádře byla schopná zajistit deterministické časování a zachycení snímku v nejkratším čase oproti implementaci využívající aplikační jádra. Deterministické časování u aplikací běžících na aplikačním jádře nebylo dodrženo ani při nejvyšší prioritě a vykonávání na izolovaném jádře.

Celkově se podařilo připravit podporu pro snadnější vývoj robotických aplikací využívající M4 jádro pro obsluhu časově kritického kódu. Práce byla pro mě velkým přínosem, protože jsem měl možnost pochopit principy až na úrovni jádra operačního systému a možnost provádět změny v různých částech celého systému.

Literatura

[1] PRU-ICSS Resources. *BeagleBoard.org - community supported open hardware computers for making* [online]. 2018 [cit. 2019-02-27]. Dostupné z: https://beagleboard.org/pru

[2] *I.MX APPLICATIONS PROCESSORS* [online]. NXP [cit. 2019-02-27]. Dostupné z: https://nxp.com/imx

[3] https://www.nxp.com/imxrt I.MX RT Series: Crossover Processor. *NXP Semiconductors / Automotive, Security, IoT* [online]. [cit. 2019-03-02]. Dostupné z: https://www.nxp.com/imxrt

[4] PRCHAL, Aleš. *Řízení robotické ruky pomocí vícejádrového hybridního procesoru* [online]. Ostrava, 2017 [cit. 2019-03-02]. Dostupné z: https://hdl.handle.net/10084/128425. Diplomová práce. Vysoká škola báňská - Technická univerzita Ostrava.

[5] *MCIMX8M-EVK: Evaluation Kit for the i.MX 8M Applications Processor* [online]. NXP [cit. 2019-02-27]. Dostupné z: https://www.nxp.com/support/developer-resources/run-time-software/i.mx-developer-resources/evaluation-kit-for-the-i.mx-8m-applications-processor:MCIMX8M-EVK

[6] *I.MX 8M SOM Starter Kit* [online]. EmCraft [cit. 2019-04-03]. Dostupné z: https://www.emcraft.com/products/868#starter-kit

[7] *PICO-IMX8M SYSTEM ON MODULE PRODUCT MANUAL (WITH NXP i.MX8M SoC)* [online]. Verze 0.1. Taiwan: TechNexion, 2019 [cit. 2019-02-27]. Dostupné z: https://cdn.sos.sk/productdata/bf/ca/bf33157a/pico-imx8mq13-r10-e08-9377.pdf

[8] *Linux kernel source tree* [online]. [cit. 2019-02-27]. Dostupné z: https://github.com/TechNexion/linux

[9] *I.MX Linux Yocto Project BSP 4.9.88-2.0.0_ga Release* [online]. TechNexion [cit. 2019-03-02]. Dostupné z: https://github.com/TechNexion/meta-edm-bsp-release

[10] GÜNTHER, Richard. *Kernel Support for miscellaneous (your favourite) Binary Formats v1.1* [online]. [cit. 2019-02-27]. Dostupné z: https://www.kernel.org/doc/Documentation/admin-guide/binfmt-misc.rst

[11] *Pico i.MX7 Development Kit for Android Things Hardware Manual.* REV B1. 2017. Dostupné také z: https://www.nxp.com/docs/en/user-guide/PICO-IMX7D-USG.pdf

[12] *Getting Started with MCUXpresso SDK i.MX 8M Quad: User's Guide.* Rev. B. 2017.

[13] *I.MX 8M Dual/8M QuadLite/8M Quad: Applications Processors Reference Manual* [online]. Revize 0. 2018 [cit. 2019-02-27].

[14] LUKÁŠ, Petr. *: RPMsg Messaging Protocol* [online]. 2016 [cit. 2019-02-27]. Dostupné z: https://github.com/OpenAMP/open-amp/wiki/RPMsg-Messaging-Protocol

[15] Virtio: Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.* ACM, 2008, **2008**(5). DOI: 10.1145/1400097.1400108.

[16] *Remote Processor Framework* [online]. [cit. 2019-03-03]. Dostupné z: https://www.kernel.org/doc/Documentation/remoteproc.txt

[17] *Cortex - M4 Devices: Generic User Guide* [online]. 2011. ARM [cit. 2019-02-27]. Dostupné z: http://infocenter.arm.com/help/topic/com.arm.doc.dui0553b/DUI0553.pdf

[18] HUSSEIN, Nur. *Randomizing structure layout* [online]. 2017 [cit. 2019-04-03]. Dostupné z: https://lwn.net/Articles/722293/

[19] KOTZIAN, Jiri. *Influence of Pin Setting on System Function and Performance* [online]. 2015 [cit. 2019-02-27]. Dostupné z: https://www.nxp.com/docs/en/application-note/AN5078.pdf

[20] *Subsystem drivers using GPIO* [online]. [cit. 2019-02-27]. Dostupné z: https://www.kernel.org/doc/Documentation/gpio/drivers-on-gpio.txt

[21] *Rosserial_tivac_tutorials: FreeRTOS Example* [online]. [cit. 2019-04-03]. Dostupné z: https://github.com/vmatos/rosserial_tivac_tutorials/tree/master/freertos123

[22] *TSL1401R-LF: 128x1 Linear sensor array with hold.* 2006. Dostupné také z: https://www.farnell.com/datasheets/315815.pdf

A Příloha

Součástí práce je elektronická příloha s následující adresářovou strukturou:

```
picopi-ros
├── linux .....změny jádra oproti TechNexion verzi tn-imx_4.9.88_2.0.0_ga-test
├── picopi8m-ros-demos ..........................ukázky periférií a použití ROS
│   ├── m4 ..........................................ukázky periférií pro M4 jádro
│   ├── linux-periph ................................ukázky periférií pro Linux
│   ├── linecam .....................ukázka pro snímání snímků na řádkové kameře
│   ├── benchmark .........porovnání časové odezvy při zasílání zpráv mezi jádry
│   ├── gpio_interrupts .......ROS ukázka využívající M4 jádro pro počítání hran
│   ├── motors ...................ROS ukázka řídící motory pomoci PWM a časovačů
│   ├── rpmsg
│   │   ├── examples ..............ukázky pro ovladač imx_rpmsg_tty a rpmsg_m4char
│   │   └── udp ............................ovladač pro zasílání zpráv z M4 po UDP
│   ├── tools ..........................................adresář nastavený v $PATH
│   │   ├── dump_rpmsg.c .............zobrazení paměti RPMsg včetně obsahu bufferů
│   │   ├── m4build ............skript pro zkompilování kódu z aktuálního adresáře
│   │   ├── m4run ..skript pro zkompilování a nahrání kódu z aktuálního adresáře
│   │   ├── m4ctl ...............skript pro ovládání a nahrávání kódu na M4 jádro
│   │   └── setup.bash ..........source skript pro nastavení proměnných prostředí
│   ├── ros_m4_demo ............................ukázka ROS komunikace mezi jádry
│   ├── rosserial_rpmsg ..........klientská rosserial knihovna využívající RPMsg
│   └── ros_m4ctl .......................ROS balíček pro ovládání M4 jádra z ROS
├── picopi8m-ros-distbuild ....................skripty pro vytvoření distribuce
│   ├── config.sh .................konfigurace distribuce (verze, balíčky, ...)
│   ├── build.sh ................................skript pro sestavení distribuce
│   ├── flash.sh ..................skript pro nahrání distribuce na minipočítač
│   └── work/files .....................................konfigurace minipočítače
├── picopi-m4sdk .................MCUXPresso SDK s vytvořenými CMake knihovnami
│   ├── tools/cmake_toolchain_files .....upravené CMake skripty včetně knihoven
│   └── template
│       ├── MIMX8MQ6xxxJZ_cm4_ram.ld ........linker skript s resource_table sekci
│       ├── board.c .....nastavení STDOUT, ochrany paměti a MemManage IRQ handler
│       └── rsc_table_rpmsg.h ..............RPMsg sekce a funkce pro inicializaci
├── picopi8m-rosclient .......Docker obraz pro využití ROS funkcionality na PC
├── rosserial .....opravená verze knihovny rosserial vykonávající se na Linuxu
└── rootkit ..........................demonstrace eskalace oprávnění z M4 jádra
```